AD-A058 434 AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OHIO F/G 9/2

SIMPL-M CODE GENERATION FOR THE INTEL 8080 MICROCOMPUTER.(U)

NOV 77 J B BLADEN

UNCLASSIFIED AFIT-CI-78-65 NL

1 OF 1

AD A058434

END

DATE FILMED

10-78

DDC

SIMPL-M Code Generation for the Intel 8080 Microcomputer

A Thesis

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

Master's thesis,

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

James B. Bladen

23 Nov[redacted] 1977

91p.

AFIT-CI-78-65

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: CI 78-65
2. GOVT ACCESSION NO.:
3. RECIPIENT'S CATALOG NUMBER:
4. TITLE (and Subtitle): "SIMPL-M Code Generation for the Intel 8080 Microcomputer"
5. TYPE OF REPORT & PERIOD COVERED: Thesis
6. PERFORMING ORG. REPORT NUMBER:
7. AUTHOR(s): Captain James B. Bladen
8. CONTRACT OR GRANT NUMBER(s):
9. PERFORMING ORGANIZATION NAME AND ADDRESS: AFIT Student at the University of Virginia, Charlottesville VA
10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS:
11. CONTROLLING OFFICE NAME AND ADDRESS: AFIT/CI WPAFB OH 45433
12. REPORT DATE: 1978
13. NUMBER OF PAGES: 80 Pages
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office):
15. SECURITY CLASS. (of this report): Unclassified
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:
16. DISTRIBUTION STATEMENT (of this Report): Approved for Public Release; Distribution Unlimited
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report):
18. SUPPLEMENTARY NOTES:

JOSEPH P. HIPPS, Major, USAF
Director of Information, AFIT
AUG 15 1978

19. KEY WORDS (Continue on reverse side if necessary and identify by block number):
20. ABSTRACT (Continue on reverse side if necessary and identify by block number):

78 08 31 003

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

APPROVAL SHEET

This thesis is submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

Author

This thesis has been read and approved by the Examining Committee:

Thesis Adviser

Accepted for the School of Engineering and Applied Science:

Dean, School of Engineering and Applied Science

ABSTRACT

Microcomputers add a new dimension to modern computers. Their small size and price make them economic for applications which seemed impractical a few years ago. However, since microcomputers are not well-suited to running large compilers, the medium of communication between the programmer and micros has traditionally been assembly language.

The computer software industry realized some time ago that the most effective means of generating a software system is by using a compiler capable of communicating clearly both with the programmer and the computer hardware. Such compilers are necessarily large, and are generally used to create machine code for the big machines they run on. However, by replacing a compiler's code generator with the code generator of a small machine, a compiler can run on the large machine and generate code for a small one, and the size limitation problem is eliminated.

This thesis presents just such a cross-compiler. The large machine is the CDC 6000 and the very capable compiler is the SIMPL compiler written by Victor R. Basili and Albert J. Turner. The microcomputer chosen as the target machine for the code generated is the Intel 8080, a well known micro and typical of the architecture and instruction capability of microcomputers. The cross-

compiler created by adding the new code generator to SIMPL is called SIMPL-M and the code generator itself is called CODGEN.

CODGEN was created using the CDC version of SIMPL-T written by John G. Perry, Jr. However, since CODGEN is written in SIMPL, and therefore is independent of the CDC hardware and software, it can easily be transported to other SIMPL packages such as the UNIVAC 1108 version. Further, CODGEN has been written with enough flexibility that it could be used as a guideline in the extension of SIMPL-M to other microcomputers.

SIMPL-M utilizes separately loaded Input/Output modules rather than a system library. In this way I/O modules can be permanently loaded into the microcomputer's PROM memory for use by SIMPL-M's external subroutine features.

SIMPL-M has been verified in that: 1) a significant but not sufficient program named HADAM has been compiled on the CDC Cyber 172, 2) a paper tape of the Intel machine code was punched, 3) the tape was loaded on the UVA Modular System Intel 8080, 4) the program executed successfully.

Table of Contents

1. Introduction

Microcomputers are a new and powerful addition to modern computing technology. They can be distinguished from other computers in that many computing functions are incorporated into one integrated circuit. This allows computing with a minimum investment of a CPU and a power supply. Adam Osborne describes a microcomputer by: "A microcomputer is a logic device. More precisely, it is an indefinite variety of logic devices, implemented on a single chip; and because of the microcomputer, logic design will never be the same again" (7, p.1-1).

A microcomputer is not very useful by itself. Data and commands must be transmitted in and out; in other words, some form of communication between the user and the microcomputer must be established.

One device that must be added by a micro user is some sort of data storage device. This can be a teletype with paper tape capability, a mass storage device such as a magnetic tape or disk drive, or one of the many forms of memory available to the micro user.

The choice of which storage device to use is generally determined by user needs and budget. Often a small system is established, and then expanded as greater needs and resources are realized.

Thus a microcomputer system is tailored by the user to his requirements, usually with little concern as to how

his system complies with other micro systems. While this flexibility is an asset which makes micros useful to many users, it puts a great demand on the programmer since each system can vary widely in configuration and capability.

One good way to program such a system is in the machine's own language since this language is tailored to the specific requirements of the computer's hardware. Many microcomputer users program small applications in machine language since it requires a minimum investment in storage space and input/output capability. However, machine language coding is tedious and impractical for large programs.

The next higher language level is assembly language.

Assembly language instructions correspond exactly with machine language instructions, but are more intelligible to the user. Most microcomputers presently in use are programmed with assemblers. Those who are familiar with microcomputer assemblers are also probably familiar with the technique of assembling a program on a large computer and then transmitting the machine code output to the micro. This could be called cross-assembling since one machine is used to generate the code which another machine executes. The term cross-compiling as used later in this text is comparable to cross-assembling.

Still a higher level language is available to the micro user. Compilers are written in languages which

are highly readable and logical to the programmer. Also compilers are generally designed to fill a user need; that is, the compiler satisfies the requirements of the user first, and then generates whatever machine code is necessary to satisfy the requirements of the machine. For this reason, a compiler may not produce machine code which is compacted to the minimum number of instructions which will do the job. The luxury of writing a program in compiler language is paid for by less than optimal machine code.

The benefits of high-level languages far outweigh the weaknesses. Some of these benefits are:

1) Compiler languages are easier to learn and use than lower languages.
2) Compiler languages are problem- rather than machine-oriented.
3) Compilers can be made to be transportable from one machine to another.
4) Good compiler languages are self-documenting and have extensive error checking capability.

But these benefits present still another problem other than less than optimal machine code. The more powerful the compiler, the larger it must be. Compilers are impractical for most micro users since they require much more storage than many machines can address. Therefore compiler execution is slow due to input and output of the memory loads required by

such large programs.

The obvious answer to the problem of executing a large program which is to produce machine code for a small computer is to implement a cross-compiler similar to the cross-assembler previously mentioned.

This thesis presents just such a cross-compiler. The compiler chosen to convert to a cross-compiler is SIMPL-T, written by V. R. Basili and A. J. Turner of the University of Maryland. They describe SIMPL-T by: "SIMPL-T is a member of a family of languages that are designed to be relatively machine independent and whose compilers are relatively transportable onto a variety of machines. It is a procedure oriented, non-block structured programming language that was designed to conform to the standards of structured programming and modular design." (1, p.v)

SIMPL-T was chosen because it is a very capable language, and yet it is "simple" in that its only data structure is the one-dimensional array. Also it allows three types of data: integer, string, and character. For a complete description of SIMPL-T see reference (1).

The new language created by converting SIMPL-T to a cross-compiler has been designated SIMPL-M (for micro). Some features of SIMPL-T such as recursive procedures and string data type have not been implemented in SIMPL-M since they were judged impractical for microcomputers. However, since the SIMPL-T compiler itself still has the potential of

providing these features, it is feasible that they can be added in the future. A complete list of SIMPL-M restrictions is listed in reference (2).

SIMPL-T, like most good compilers, has a modular design. It is made up of the following basic modules; 1) Scanner, 2) Parser, 3) Code Generator. The SIMPL-T compiler can be made to generate machine code for virtually any machine by first implementing it on a host machine, and then replacing module 3 with a code generator for the new machine. This requires no alteration to the existing compiler whatsoever beyond replacing the code generator.

SIMPL-T was originally implemented on the UNIVAC 1108 computer. Since this project was accomplished at the University of Maryland, the first obstacle was to transfer SIMPL-T to UVA's CDC Cyber 172. Fortunately, John Perry of Dahlgren Labs, Va. has already implemented SIMPL-T on the CDC 6000 (9) and he provided a copy of his program for this project. This compiler has been brought up on the UVA Cyber and can be run using the procedure outlined in Appendix V.

The next and greatest obstacle was to decipher the documentation of Perry's compiler in order to determine what information the SIMPL-T compiler modules pass to the code generator module. A large part of this paper is dedicated to the documentation of how SIMPL-T communicates with its code generator in order to remove this obstacle.

The choice of which microcomputer to choose as a target machine for the cross-compiler was easy. The Tntel 8080 is representative of the capabilities of many micros and has been used as a standard for many newer ones. Adam Osborne (8, p. 4-1) states that; "The 8080A is the most widely known of the microcomputers (described in this chapter); as such, it becomes the frame of reference in many people's minds as to what a microcomputer should be. ...the 8080A was designed...at a time when no definable microcomputer user public had established itself...

"The success of this microcomputer is due either to the farsighted genius of its designers, or to the fact that the power of most microcomputers so overwhelms the needs of microcomputer applications, the CPU design becomes almost irrelevant when compared to product costs and product availability." (8, p. 4-1)

Although SIMPL-M has been implemented for the Intel 8080 only, it is intended to be flexible enough to be applicable to other micros.

The SIMPL-M cross-compiler is the SIMPL-T compiler modified with a replacement code generator module CODGEN.

The CODGEN module is itself written in SIMPL-T and is compiled separately from the rest of the SIMPL-T package. Once it has been compiled and assembled into CDC machine code, it is merged into the rest of the SIMPL-T package at load time. The resulting absolute binary core image is catalogued as SIMPL-M. Due to the large size of the total package, six levels of overlays are required for the final load. A complete description of the process to compile, overlay, load, and catalog SIMPL-M is given in Appendix IV.

This thesis is intended to document the code generator CODGEN as it is presently written, to act as a guide in future changes to CODGEN, and to clarify the communication process between the SIMPL-T compiler and its code generator. A user's manual for SIMPL-M has been written which will be changed as SIMPL-M is updated (refeŕence 2).

2. Communication between SIMPL-T and the SIMPL-M Code Generator - CODGEN

As we have noted, SIMPL-T is made up of three basic modules; 1) Scanner, 2) Parser, and 3) Code Generator. Due to the large size of the compiler, these modules are not resident in the Cyber memory all at once. Instead, a driver program and a symbol table array stay fixed in memory, and the modules are swapped in and out in sequence. Figure 1 is a conceptual flow chart of the SIMPL-M system. Some of the compiler's features have been omitted or altered for clarity.

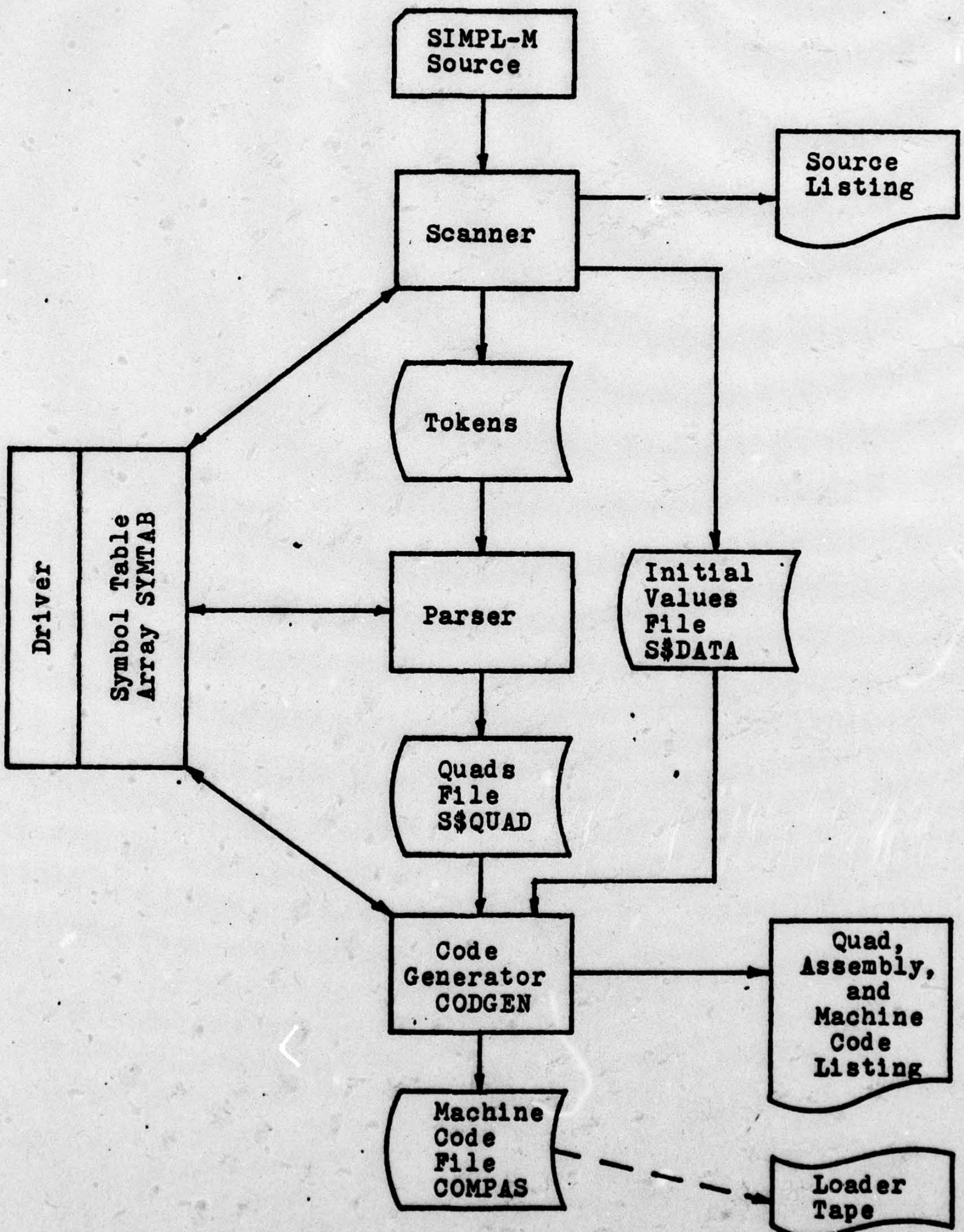


Figure 1. SIMPL-M Conceptual Flow Chart

The scanner examines the entire source deck and passes the input to the parser in the form of tokens. The parser analyzes the tokens and generates quads which it passes to the code generator. These quads, the initial data file, and the symbol table array contain all the information necessary for the code generator to produce machine code for the target machine (the Intel 8080 in the case of SIMPL-M).

As seen in Figure 1, CODGEN receives input from the S$QUAD file, the SYMTAB array, and the S$DATA file. We will discuss each in turn.

2.1 Quad File S$QUAD

SIMPL-T passes information to the code generator in an intermediate language called quads. When the code generator takes over, all the quads have been defined and have been stored in the external file S$QUAD. Each quad has an ID number followed by optional integer values according to the number of quad parameters required. Array NQ contains a code number which signals PROC NQUAD (VIII, 603)* to read zero or more quad parameters.

---

*Where appropriate, specific procedures in this text will be followed by the Appendix number of a listing and a line number.

2.1.1 Quad Format

A quad that would represent an operator, such as +, has the following sequence in S$QUAD:

ID
AFLAG
A
BFLAG
B
RFLAG
R

Where ID is the number 7; AFLAG, BFLAG, and RFLAG tell the type of entry to follow, and A, B, and R fields are either 1) pointer to SYMTAB, 2) a temporary number, 3) an immediate value.

The quad flags, AFLAG, BFLAG, and RFLAG must be logically anded with the following values to determine how a corresponding field should be handled:

Quad Mask	Type
1	Temporary
2	SYMTAB Array Pointer
3	Immediate

Unfortunately, these masks do not define all possibilities for quad values. A complete breakdown can be obtained using

the algorithm defined by FUNC LABEL (VIII, 775).

2.1.2 Quad Interpretation

The quads themselves can be regarded as macro instructions. For example, the statement

DOG := (CAT + RAT) - LOST

generates the quads:

ID	A	B	R
LINE	1		
+	CAT	RAT	TEMP1
-	TEMP1	LOST	TEMP2
:=	TEMP2		DOG

Appendix III lists typical quad sequences for SIMPL-M statements.

2.2 Symbol Table Array SYMTAB

The symbol table SYMTAB is a collection of interrelated arrays maintained by the SIMPL-T compiler. All identifier names, constant values, PROC (procedure) names, local variable names, parameter names, intrinsic names, and key words are contained in these arrays.

Discussion of the symbol table will be limited to only those parts which require referencing by CODGEN. A more exhaustive discussion of the symbol table is contained in reference (9).

The symbol table entries consist of three or more words which are formatted after the UNIVAC 1108 36-bit word. On the CDC Cyber, this means the least significant 18 bits form the right half-word, the next 18 bits form the left half-word, and the rest of the CDC 60-bit word is all zeroes:

24 Zeroes	Left Half	Right Half
59 36	35 18	17 Ø

This configuration leaves 24 unused bits in the CDC word; however, by keeping the original format, the compiler retains its portability to machines with smaller word size.

Entries of similar type in SYMTAB are linked by pointers which make up a type chain. The entry in each chain is passed to CODGEN via an external pointer. For example, all constants are linked by a chain and the first entry is pointed to by FCPTR. The next three sections discuss the three type chains used by CODGEN.

2.2.1 <u>Uninitialized Globals - PROC ALLOCGLOBALS</u>

The first symbol table type examined in CODGEN is uninitialized globals. All globals are linked in SYMTAB by the global chain. PROC ALLOCGLOBALS (VIII, 617) examines the entries in the global chain and dedicates a memory location to uninitialized globals only (initialized globals are dealt with in PROC ALLOCDATA (VIII, 663)).

As PROC ALLOCGLOBALS traverses the global chain, it

checks the description entry of each global; and when it finds an uninitialized entry, it allocates a memory location and puts the address in the right half of the I3 word of the SYMTAB entry. See Figure 2.

2.2.2 Constants - PROC CHECKCONSTANTS

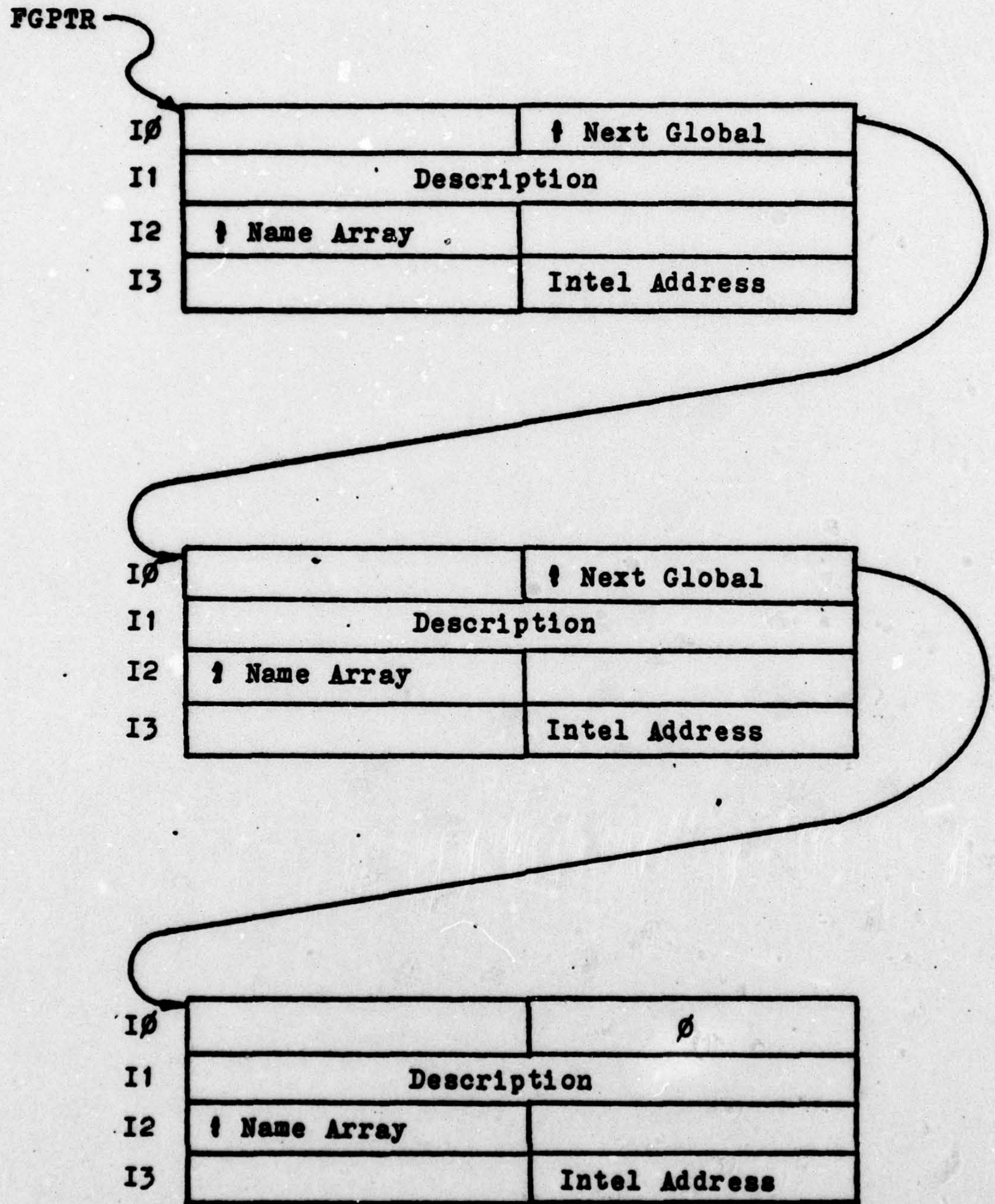
PROC CHECKCONSTANTS (VIII, 650) traverses the constant chain and checks the range of all constants. Since the Intel 8080 has an 8-bit word, and the most significant bit must be treated as a sign bit to uniquely identify a number, the range checked is -2^7 to 2^7-1 or -128 to 127. Numbers as large as 255 can be input, but will generate a diagnostic warning. See figure 3.

2.2.3 Parameters and Locals - PROC IDPROC

When a PROC (procedure) quad is encountered by CODGEN, its parameters and local variables must be given memory locations. These are located in the symbol table and are connected by the parameter/local chain. The entry pointer to the parameter/local chain is in the right half of the IO word of the PROC symbol table entry. See Figure 4.

2.2.4 More on SYMTAB

The name of any symbol table entry is retrieved with the external function NAME (VIII, 393). NAME has two parameters, the T1 address of the symbol table entry and the number 1. The value returned is a string. NAME must be



<u>Figure 2. The Global Chain</u>

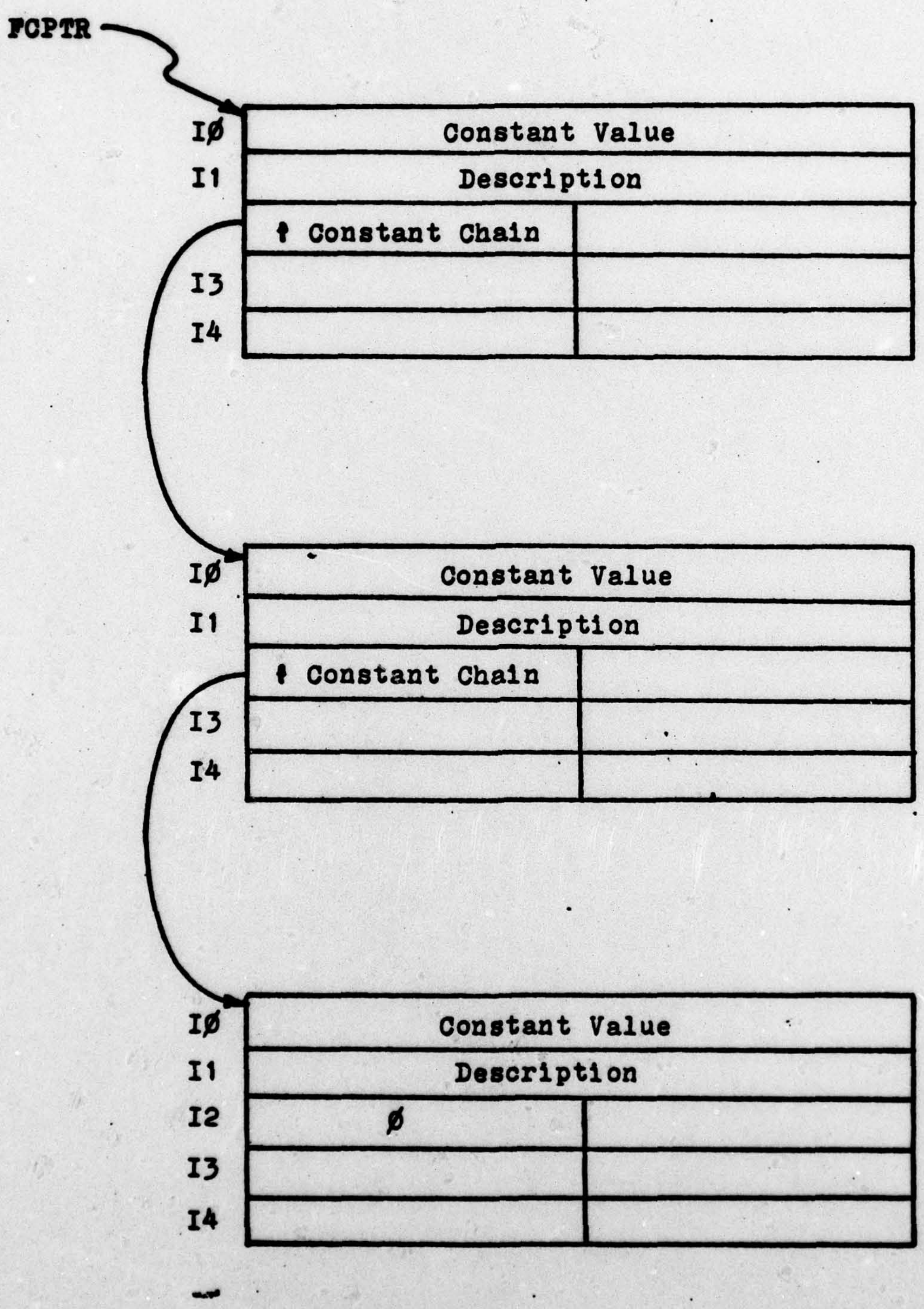


Figure 3. The Constant Chain

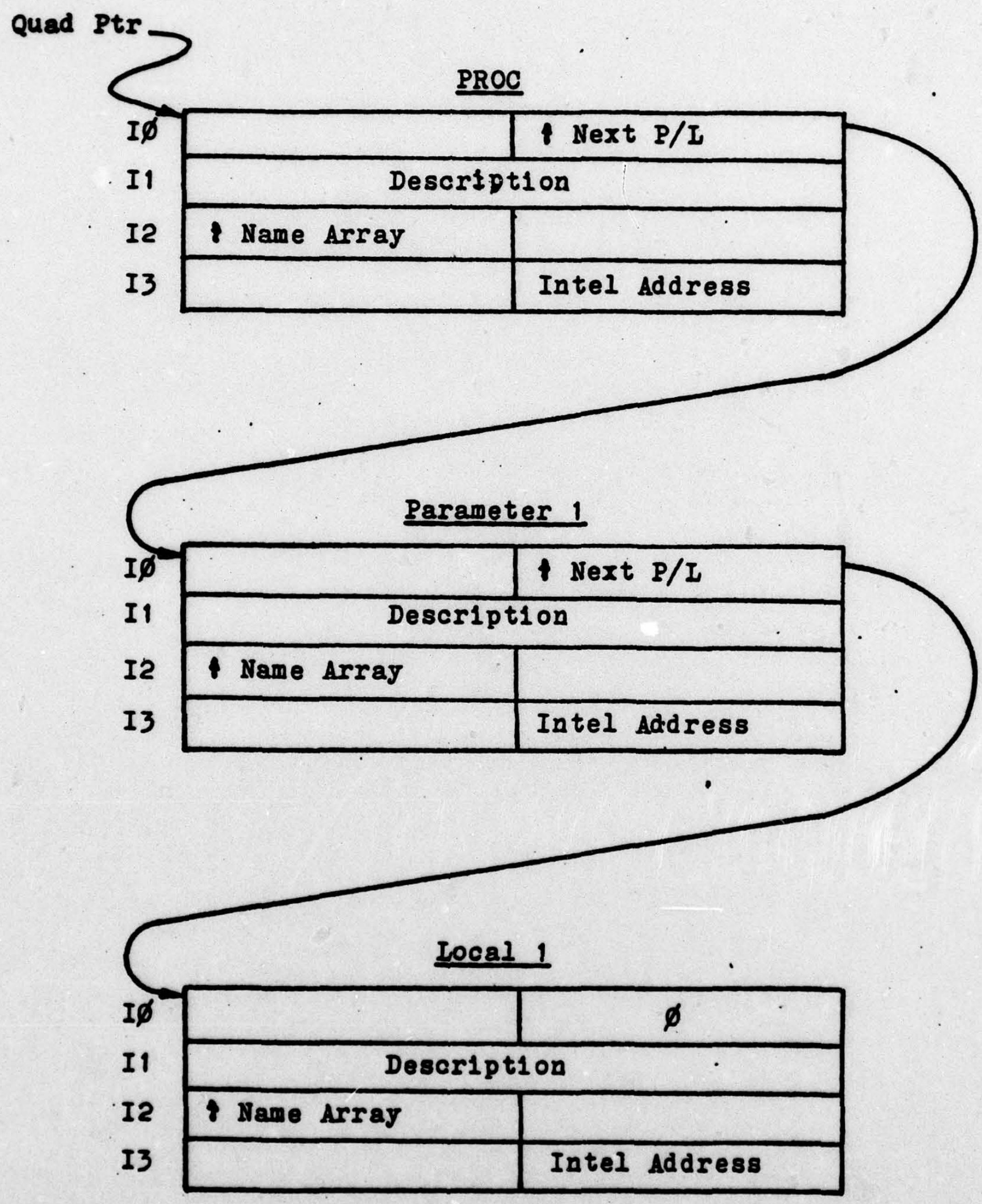


Figure 4. The Parameter / Local Chain

declared as:

```
EXT STRING FUNC(INT,INT)   (VIII, 12)
```

The type of any symbol table entry is obtained by performing a logical AND with a bit mask and the SYMTAB description word. PROC LABEL (VIII, 775) demonstrates the use of the following binary masks:

SYMTAB Mask	Type
1	Integer
1111	FUNC
1Z13*	Array
1Z19	Constant
1Z31	Initialized (S$DATA entry)
1Z22	Parameter
1Z25	Reference Parameter

2.3 File S$DATA - Initialized Globals - PROC ALLOCDATA

Initial values assigned to globals are stored in file S$DATA. PROC ALLOCDATA (VIII, 663) searches this file and stores initial values in the memory. The memory address is then stored in the SYMTAB entry for the globals If ALLOCDATA encounters an initial value greater than the maximum allowable integer (presently 127), the initial value is

---

*As in SIMPL-T, the NZnn notation denotes there are decimal nn zeroes following the number N.

assumed to be an address. In this case, the value itself is stored in the SYMTAB address field, and no memory location is allocated. This feature is necessary for input/output as it is presently defined.

Figure 5 shows the S$DATA representation of integers and Figure 6 shows the representation of arrays.

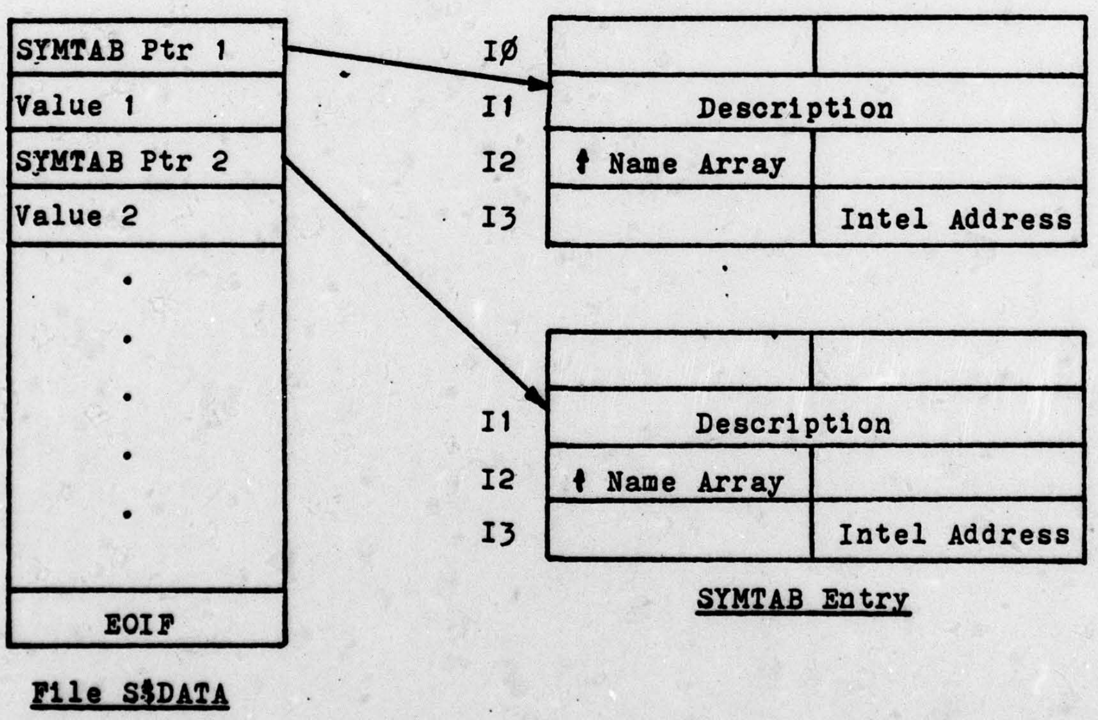
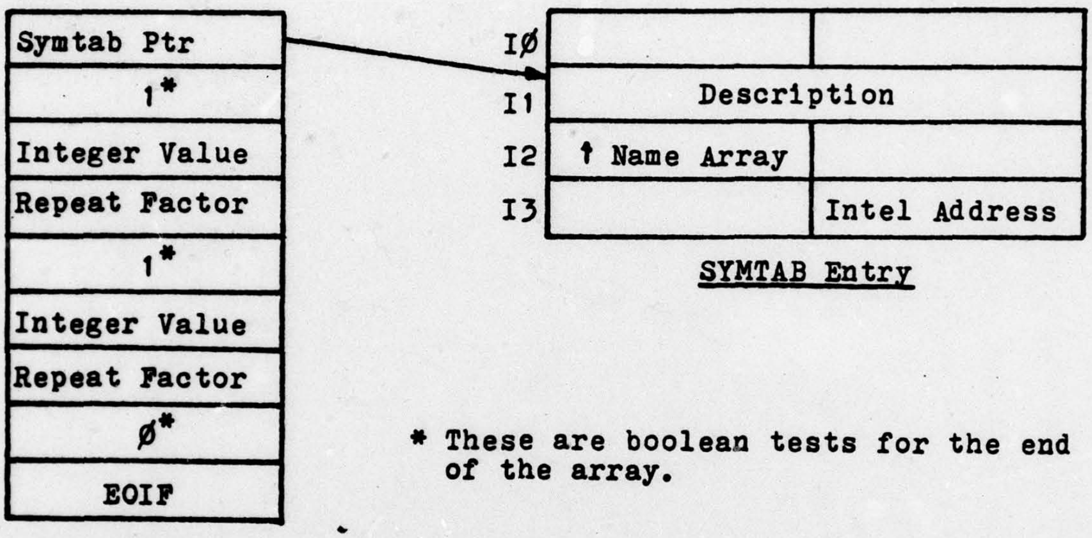


Figure 5. File S$DATA Integers

Format:



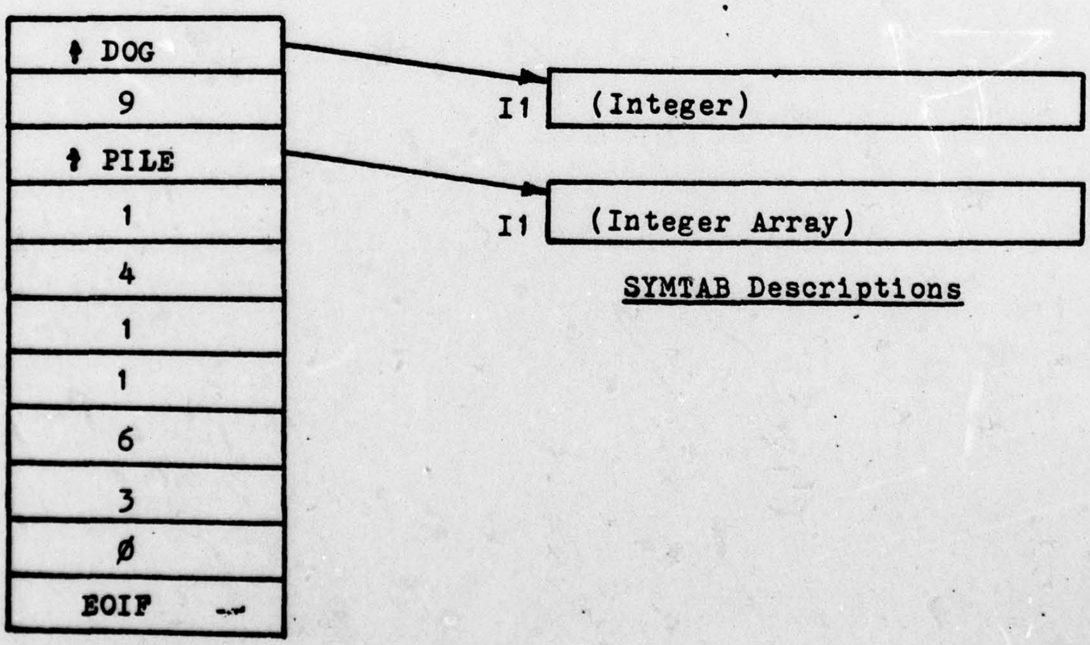
File S$DATA

SYMTAB Entry

* These are boolean tests for the end of the array.

Example:

```
INT DOG = 9
INT ARRAY PILE(6) = (4, 6(3))
```



File S$DATA

SYMTAB Descriptions

Figure 6. File S$DATA Arrays

3. Machine Code File COMPAS

Intel 8080 machine code is output to external file COMPAS in string format. Each call to external PROC WRITEC outputs a string record to COMPAS (Appendix I, next to last page), and puts an EOR mark at the end. The header LDR and delimiter ; are also written to COMPAS as required by the UVA Modular System (2).

3.1 Sequential Code - PROC WRITCODE

Intel machine code for each instruction is included in the assembly code string declarations (VII ., 47). During code generation, immediate values and addresses are concatenated to this and the resulting string is passed to PROC WRITCODE (VII , 843). WRITCODE considers all input to it sequential and increments the memory pointer accordingly. The machine code is removed from the input string as a substring, and this is concatenated to a buffer string (CODEBUF). When the buffer becomes full, or it is to be flushed (see section 4.2), the buffer is output to file COMPAS.

3.2 Non-Sequential Code - PROC WRITBACK

Forward references in CODGEN are kept in the SAVE stack. As the values of these references becomes known, they are written out of sequence by PROC WRITBACK (VII , 882). WRITBACK always checks to see if the code buffer is empty before it outputs to prevent WRITCODE from writing a dummy

value over the forward reference. If the buffer is non-empty then it is flushed before WRITBACK outputs the reference.

4. Code Generation Messages

Two types of printed output are available from CODGEN; optional and required. Optional output is specified on the compiler execution card as an L or Q parameter (see reference 2, compiler options).

4.1 Required Output

If a start load address was specified in the program then it is printed out first (the default start load address is zero). Next the maximum address used by the compilation is written, and finally a start execution address is written. These three parameters are sufficient to define a compiled module for external use.

The only other required messages are errors and they are delimited by:

>>>>ERROR<<<< error message >>>>ERROR<<<<

Error messages contain a source line number where applicable.

4.2 Optional Output

Specifying L as a compiler option generates Intel 8080 assembly code and memory addresses. No labels are generated since actual addresses are listed. Some clarifying comments have been added using * as a delimiter.

Specifying Q as a compiler option prints out the quads in comment format. PROC LABEL (VII , 775) determines what type A, B, and R fields are by examining the appropriate flags and SYMTAB entries.

5. Input/Output and Start Load Capabilities

There is no program library available for the present version of SIMPL-M. Instead, there are facilities to call external pre-loaded routines such as the ones usually available in the PROM monitor or loader of a microcomputer. These routines are called by PROC EXTPROC and FUNC EXTFUNC (see reference 2, I/O). EXTPROC (VII , 395) and EXTFUNC (VII , 393) are reserved words within CODGEN only. An external procedure's start address is passed as a parameter so that any number of pre-loaded routines may be utilized. Both EXTPROC and EXTFUNC require that the subroutine argument be passed in the accumulator. See section 7 for a more thorough explanation of SIMPL-M I/O.

STARTLOAD (VII , 688) is another CODGEN reserved word. When it is encountered, the memory pointer is set to its declared value. The default is zero, and if STARTLOAD is declared it should be the first statement in a program.

6. Optimization and Verification

Some code optimization has been utilized in CODGEN. This is accomplished with a look-ahead technique. The next quad is always available so that a temporary is not pushed

into the Intel stack if the A field of the next quad is a temporary. Thus the temporary is held in the accumulator to avoid a PUSH command followed immediately by the POP command.

The SIMPL-M compiler has been tested only in that: 1) a significant but not sufficient program was compiled on the Cyber 172, 2) a paper tape of the Intel machine code was punched, 3) the tape was loaded on the UVA Modular System Intel 8080, 4) the program executed successfully. The test program HADAM (I) multiplies the 8x8 Hadamard matrix times an input vector, and outputs the result. Since the matrix was known in advance to have all entries either +1 or -1, no actual multiplication is performed.

This test program uses many of the capabilities of the SIMPL-M compiler, but not all of them. For this reason, SIMPL-M cannot at this time be considered verified, but its verification will be completed when time permits.

7. Conclusion

One of the most important assumptions underlying this thesis is that a 7-bit integer is reasonable for most applications of the Intel 8080. The 8080 has double word instructions so that the word size could and should be extended to 15 bits. However, if all integers in SIMPL-M are to be treated as 15 bits, then the efficiency of the resultant machine code will drop considerably since 16-bit manipulations are cumbersome on this machine. For this reason, the addition

of extended precision to SIMPL-M must be treated as an optional rather than a mandatory feature since limited memory size will probably continue to be important to 8080 users.

The second important assumption is that since micro users have widely varied I/O resources, exhaustive library facilities are out of the question. SIMPL-M does not have a set of library routines which are loaded with the program or which are disk-resident awaiting a call from the system loader. Instead, this thesis contends and demonstrates that I/O can be handled by accessing preloaded routines - particularly those routines which are permanently resident in PROM. Most existing systems have a monitor with its I/O routines resident in PROM. By using these routines, they need not be loaded each time they are used, plus each user can tailor his I/O to his requirements and still interface with SIMPL-M.

The best way to demonstrate how SIMPL-M performs I/O using pre-loaded routines is with an example. The program HADAM, which was previously mentioned, is a good example of SIMPL-M I/O. HADAM was written for the UVA Modular System which has very basic I/O subroutines and, therefore much of the program is dedicated to handling I/O. More sophisticated I/O could easily be accomplished by extending the existing PROM routines. The notation (ln ___) used throughout this section refers to line numbers in the HADAM

routine in Appendix I.

The UVA Modular System has a PROM resident monitor in the first 1K of address space. Note that HADAM's program load starts at ØØØØ (ln 6). Included in the monitor are some limited I/O routines. The start address of each of these routines is declared in HADAM from ln 12 through ln 18 as integers. As mentioned in Section 2.3, any initial value greater than 127 decimal is assumed to be an address.

The UVA monitor I/O routines assume the subroutine argument is always passed in the accumulator, and the EXTPROC and EXTFUNC facilities (see Section 5) also make this assumption. EXTPROC and EXTFUNC are key words within CODGEN which signal the code generator to perform a subroutine call to the address passed as the first parameter. In the case of EXTPROC, before the call is made, the accumulator is loaded with the value of the second argument. In the case of EXTFUNC, the call is made and the returned accumulator value becomes the returned function value. Thus ln 92 loads the accumulator with an ASCII minus and performs a subroutine call to location H!00A6! which has a preloaded character -- print routine. Likewise, ln 71 performs a function call to location H!ØØD9! which has a preloaded character-read routine. The value returned in the 8080 accumulator then becomes the function argument and is compared wich ASCII minus in the IF statement (actual assembly and machine code for this statement can be found under

line 71 in the assembly listing in Appendix 1).

By filling an initialized array (ln 43) with ASCII characters and inserting the character-print call in a WHILE loop (ln 59), string output can be accomplished even with the existing primitive constructs. Printing the 8X8 Hadamard matrix is relatively simple by nesting PRINTVECTOR (ln 85) in a WHILE loop (ln 106) and using the preloaded print-two-hex-characters routine THCHO(ln 97).

SIMPL-M's interactive capability is demonstrated by QUERY (ln 113). When QUERY is called (ln 146), it asks if the previous output is OK and then inputs a character (ln 116). QUERY returns TRUE if the input is ASCII N for NO, or FALSE if the input is otherwise. The output created by HADAM as it executes on the UVA Modular System Intel 8080 is the last page of Appendix 1.

SIMPL-M's capability to use pre-loaded subroutines will be expanded as the need arises. The next obvious addition will be a call which passes an address and an integer as arguments so that a preloaded string output routine will have a string start address and a string length to work with. Once all combinations of external routine execution requirements have been defined and implemented in SIMPL-M, any new or existing I/O facilities can be handled with SIMPL-M by accessing user defined procedures.

Bibliography

(1) Basili, V. R., and Turner, A. J. 1976. SIMPL-T: A Structured Programming Language. Palladin House Publishers, Geneva, Ill.

(2) Bladen, J. B., Basili, V. R., and Turner, A. J. 1977. SIMPL-M: A Structured Programming Language for Microcomputers. Technical Report. The University of Virginia, Charlottesville, Va.

(3) Cyber Common Utilities Reference Manual. 1975. Control Data Corporation, St. Paul, Minn.

(4) Intel 8080 Assembly Language Programming Manual. 1976. Intel Corporation, Santa Clara, Ca.

(5) McDonald, Wesley E. UVA Modular Microcomputer Systems Basic Monitor. 1976. Unpublished Paper. The University of Virginia, Charlottesville, Va.

(6) NOS/BE 1 Reference Manual, Cyber 170 Series. 1977. Control Data Corporation, St. Paul, Minn.

(7) Osborne, Adam. 1976. An Introduction to Microcomputers. Volume I : Basic Concepts. Osborne and Associates, Inc. Berkeley, Ca.

(8) Osborne, Adam. 1976. An Introduction to Microcomputers Volume II : Some Real Products. Osborne and Associates, Inc. Berkeley, Ca.

(9) Perry, J. G., Jr. 1976. CDC 6000 SIMPL-T Compiler Internal Documentation. Unpublished Masters Thesis. The University of Maryland, College Park, Md.

(10) Rill, J. K. and Stager, T. W. 1974. PUNPAPR - Punch Information on Paper Tape. LIB - 0023, Computing Center, The University of Virginia, Charlottesville, Va.

(11) UPDATE Reference Manual. 1975. Control Data Corporation, St. Paul, Minn.

Appendix I. HADAM Listing

```
/*THIS PROGRAM DEMONSTRATES PRE-LOADED I/O ROUTINES IN SIMPL-M.*/
/*THE PROGRAM INPUTS A VECTOR FROM THE KEYBOARD, PERFORMS THE HADAMARD*/
/*TRANSFORM ON THE VECTOR, AND PRINTS OUT THE RESULT*/

INT STARTLOAD = H↑0400↑

EXT INT FUNC EXTFUNC(INT)
EXT PROC EXTPROC(INT,INT)

/*EXTERNAL SUBROUTINE ADDRESS DECLARATIONS*/
INT     CRLF = H↑0181↑, /*PRINT CARRIAGE RETURN, LINE FEED*/
        CHOUT = H↑00A6↑, /*PRINT ONE ASCII CHARACTER*/
        CHIN = H↑00D9↑, /*INPUT ONE ASCII CHARACTER*/
        DIGOUT = H↑0143↑, /*OUTPUT ONE HEX CHARACTER*/
        GETA = H↑0167↑, /*INPUT ONE HEX CHARACTER*/
        THCHI = H↑0153↑, /*INPUT TWO HEX CHARACTERS*/
        THCHO = H↑0132↑  /*OUTPUT TWO HEX CHARACTERS*/

/*ASCII CHARACTERS*/
INT     MINUS = H↑2D↑,  BLANK = H↑20↑,  ZERO = H↑30↑,
        COMMA = H↑2C↑, NCHAR = 78

/*OTHER GLOBALS*/
INT     N = 8, /*SIZE OF MATRIX*/
        TRUE = 1,       FALSE = 0

/*HADAMARD MATRIX*/
INT ARRAY H(64) = ( 1, 1, 1, 1, 1, 1, 1, 1,
                    1,-1, 1,-1, 1,-1, 1,-1,
                    1, 1,-1,-1, 1, 1,-1,-1,
                    1,-1,-1, 1, 1,-1,-1, 1,
                    1, 1, 1, 1,-1,-1,-1,-1,
                    1,-1, 1,-1,-1, 1,-1, 1,
                    1, 1,-1,-1,-1,-1, 1, 1,
                    1,-1,-1, 1,-1, 1, 1,-1 )

/*INPUT DATA AND OUTPUT RESULT VECTORS*/
INT ARRAY DATA(8), RESULT(8)

/*ASCII MESSAGE ARRAYS*/
                    /*CR,LF,LF,LF,LF, H, A, D, A, M, A, R, D,  , M, A,*/
INT ARRAY HEADER(24)=(13,10,10,10,10,72,65,68,65,77,65,82,68,32,77,65,
                    /* T, R, I, X,CR,LF,LF,LF*/
                     84,82,73,88,13,10,10,10),
               /*CR,LF,LF, O, K,QS,  , (, N, =, N, O, ),  */
          OK(14)=(13,10,10,79,75,63,32,40,78,61,78,79,41,32),
                    /*CR,LF,LF, I, N, P, U, T,  , V, E, C, T, O, R,CR,*/
          INMESS(18)=(13,10,10,73,78,80,85,84,32,86,69,67,84,79,82,13,
                    /*LF,LF*/
                     10,10),
                    /*CR,LF,LF, R, E, S, U, L, T,CR,LF,LF*/
          OUTMESS(12) = (13,10,10,82,69,83,85,76,84,13,10,10)
```

```
              PROC PRINT(INT ARRAY OUT, INT LOUT)
                INT LVAR
   1   1        LVAR := 0
   2   1        WHILE LVAR <> LOUT DO
   3   2            CALL EXTPROC(CHOUT,OUT(LVAR))
   4   2            LVAR := LVAR + 1
                END/*WHILE*/
   5   1      RETURN

              PROC INVECTOR(INT ARRAY VECTOR)
                /*INPUT REQUIRED IS  1) + OR - OR BLANK  2) 2 HEX CHAR NUMBER  */
                INT CNT,COL,COMPL,COM,LCL
   6   1        CALL PRINT(INMESS,18)
   7   1        CNT := N  COL := 0
   9   1        WHILE CNT DO
  10   2            COM := 0  COMPL := FALSE
  12   2            IF EXTFUNC(CHIN) = MINUS THEN
  13   3                COMPL := TRUE
                    END
  14   2            LCL := EXTFUNC(THCHI)
  15   2            IF LCL>127 THEN RETURN END
  17   2            IF COMPL THEN LCL := (.C.LCL) + 1 /*2S COMPLEMENT*/
  19   3                COMPL := FALSE
                    END
  20   2            VECTOR(COL) := LCL
  21   2            COL := COL + 1  CNT := CNT - 1
  23   2            IF CNT THEN CALL EXTPROC(CHOUT,COMMA) END
                END/*WHILE*/
  25   1      RETURN

              PROC PRINTVECTOR(INT ARRAY VECTOR, INT PTR)
                INT CNT,LCL
  26   1        CNT := N
  27   1        CALL EXTPROC(CRLF,0)
  28   1        WHILE CNT DO
  29   2            LCL := VECTOR(PTR)
  30   2            IF LCL.A.H'80' THEN
  31   3                CALL EXTPROC(CHOUT,MINUS)
  32   3                LCL := (.C.LCL) + 1 /*2S COMPLEMENT*/
                    ELSE
  33   3                CALL EXTPROC(CHOUT,BLANK)
                    END
  34   2            CALL EXTPROC(THCHO,LCL)
  35   2            IF CNT THEN CALL EXTPROC(CHOUT,COMMA) END
  37   2            CNT := CNT - 1  PTR := PTR + 1
                END/*WHILE*/
  39   1      RETURN

              PROC PRINTMATRIX
                INT  PTR, RCNT
  40   1        RCNT := N  PTR := 0
  42   1        WHILE RCNT DO
  43   2            CALL PRINTVECTOR(M,PTR)
  44   2            RCNT := RCNT - 1
  45   2            PTR := PTR + N
                END/*WHILE*/
  46   1      RETURN
```

```
  48   1      IF EXTFUNC(CHIN) = NCHAR THEN RETURN(FALSE)
  50   2      ELSE RETURN(TRUE)
              END

           PROC TRANSFORM(INT ARRAY DATAV, INT ARRAY RESULTV)
              INT RCNT,ROW,CCNT,LCL,PTR,COL
  51   1      RCNT := N  ROW := 0  PTR := 0
  54   1      WHILE RCNT DO
  55   2          LCL := 0  CCNT := N  COL := 0
  58   2          WHILE CCNT DO
  59   3              IF H(PTR) = -1 THEN
  60   4                  LCL := LCL - DATAV(COL)
                      ELSE
  61   4                  LCL := LCL + DATAV(COL)
                      END
  62   3              CCNT := CCNT - 1
  63   3              PTR := PTR + 1  COL := COL + 1
                  END/*WHILE*/
  65   2          RESULTV(ROW) := LCL
  66   2          RCNT := RCNT - 1  ROW := ROW + 1
              END/*WHILE*/
  68   1  RETURN


           PROC HADAMARD
              /*IMAGE DATA PROCESSING BY HADAMARD TRANSFORM*/
  69   1      CALL PRINT(HEADER,24)
  70   1      CALL PRINTMATRIX
  71   1      WHILE 1 DO
  72   2          CALL INVECTOR(DATA)
  73   2          CALL PRINTVECTOR(DATA,0)
  74   2          IF QUERY THEN /*INPUT IS CORRECT*/
  75   3              CALL TRANSFORM(DATA,RESULT)
  76   3              CALL PRINT(OUTMESS,12)
  77   3              CALL PRINTVECTOR(RESULT,0)
                  END
              END/*WHILE*/
           START HADAMARD
```

77 STATEMENTS IN PROGRAM

1 FUNCTION, 6 PROCEDURES
AVERAGE STATEMENTS PER PROC/FUNC: 11.0

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

*******INTEL 8080 ASSEMBLY AND MACHINE CODE*******

```
       * DECLARATION OF INITIALIZED GLOBALS
START  LOAD AT 0400
       * ADDRESS CRLF
       * ADDRESS CHOUT
       * ADDRESS CHIN
       * ADDRESS DIGOUT
       * ADDRESS GETA
       * ADDRESS THCHI
       * ADDRESS THCHO
       * MINUS
0400   EQU          2D
       * BLANK
0401   EQU          20
       * ZERO
0402   EQU          30
       * COMMA
0403   EQU          2C
       * NCHAR
0404   EQU          4E
       * N
0405   EQU          08
       * TRUE
0406   EQU          01
       * FALSE
0407   EQU          00
       * ARRAY H STARTS AT 0408
       * ARRAY HEADER STARTS AT 0448
       * ARRAY OK STARTS AT 0460
       * ARRAY INMESS STARTS AT 046E
       * ARRAY OUTMESS STARTS AT 0480
       * DECLARATION OF NON-INITIALIZED GLOBALS
       * EXTFUNC
048C   EQU          00
       * ARRAY DATA STARTS AT 048D
       * ARRAY RESULT STARTS AT 0495
WARNING - CONSTANT OUT OF RANGE = 128
       * LINE   55
       * PROC   PRINT

       **** PROC PRINT ****
       * LOCAL LVAR
0490   EQU          00
       * VAL PARM LOUT
049E   EQU          00
       * REF PARM OUT
049F   EQU          0000
       * SAVE RETURN ADDR
04A1   POP  D       D1
       * PUT VALUES IN PARMS
04A2   POP  B       C1
04A3   LXI  H       21 049E
04A6   MOV  M,C     71
04A7   POP  H       E1
04A8   SHLD ADR     22 049F
       * PUT RETURN ADDR BACK ON STACK
04AB   PUSH D       D5
       * LINE   57
       * :=     0        LVAR
04AC   MVI  A       3E 00
04AE   LXI  H       21 049D
04B1   MOV  M,A     77
       * LINE   58
       * WHILE  0
       * <>         LVAR LOUT TEMP1
04B2   LXI  H       21 049D
04B5   MOV  A,M     7E
04B6   LXI  H       21 049E
04B9   MOV  D,M     56
04BA   SUB  D       92
04BB   JZ   ADR     CA 04C0
04BE   MVI  A       3E 01
       * WHTEST TEMP1
04C0   ORA  A       B7
04C1   JZ   ADR     CA AAAA
       * LINE   59
       * CALL   EXTPROC        0
       * PARM   CHCUT          0
       * ARYLOC OUT LVAR TEMP1
04C4   MVI  B       06 00
04C6   LXI  H       21 049D
04C9   MOV  C,M     4E
04CA   LHLD ADR     2A 049F
04CD   DAD  B       09
04CE   MOV  C,M     4E
04CF   PUSH B       C5
       * PARM   TEMP1          0
       * PARM ALREADY ON STACK
       * ENDPRM
04D0   POP  B       C1
04D1   MOV  A,C     79
04D2   CALL ADR     CD 00A6
       * LINE   60
       * +          LVAR 1 TEMP1
04D5   LXI  H       21 049D
04D8   MOV  A,M     7E
04D9   MVI  D       16 01
04DB   ADD  D       82
       * :=         TEMP1      LVAR
04DC   LXI  H       21 049D
04DF   MOV  M,A     77
       * LINE   61
       * ENDWH

       * LOAD OUT OF SEQUENCE
04C2   EQU              04E3
04E0   JMP  ADR     C3 04B2
       * LINE   62
       * RETURN 0
04E3   RETURN       C9
       * LINE   64
       * ENDPRC
       * PROC   INVECTOR

       **** PROC INVECTOR ****
       * LOCAL LCL
04E4   EQU              00
       * LOCAL COM
04E5   EQU              00
       * LOCAL COMPL
04E6   EQU              00
```

```
04E8  EQU              00
      * REF PARM VECTOR
04E9  EQU              0000
      * SAVE RETURN ADDR
04EB  POP  D     D1
      * PUT VALUES IN PARMS
04EC  POP  H     E1
04ED  SHLD ADR   22 04E9
      * PUT RETURN ADDR BACK ON STACK
04F0  PUSH D     D5
      * LINE   67
      * CALL   PRINT          0
      * PARM   INMESS          0
04F1  LXI  H     21 046E
04F4  PUSH H     E5
      * PARM   18          0
04F5  MVI  C     0E 12
04F7  PUSH B     C5
      * ENDPRM
04F8  CALL ADR   CD 04A1
      * LINE   68
      * :=     N          CNT
04FB  LXI  H     21 0405
04FE  MOV  A,M   7E
04FF  LXI  H     21 04E8
0502  MOV  M,A   77
      * :=     0          COL
0503  MVI  A     3E 00
0505  LXI  H     21 04E7
0508  MOV  M,A   77
      * LINE   69
      * WHILE  0
      * WHTEST CNT
0509  LXI  H     21 04E8
050C  MOV  A,M   7E
050D  ORA  A     B7
050E  JZ   ADR   CA AAAA
      * LINE   70
      * :=     0          COM
0511  MVI  A     3E 00
0513  LXI  H     21 04E5
0516  MOV  M,A   77
      * :=     FALSE          COMPL
0517  LXI  H     21 0407
051A  MOV  A,M   7E
051B  LXI  H     21 04E6
051E  MOV  M,A   77
      * LINE   71
      * CALL   EXTFUNC          TEMP1
      * PARM   CHIN          0
051F  CALL ADR   CD 0009
      * ENDPRM
      * =      TEMP1 MINUS TEMP2
0522  LXI  H     21 0400
0525  MOV  D,M   56
0526  SUB  D     92
0527  JZ   ADR   CA 052E
052A  XRA  A     AF
052B  JMP  ADR   C3 0530
052E  MVI  A     3E 01
      * IF     TEMP2
0530  ORA  A     B7
0531  JZ   ADR   CA AAAA
```

```
0537  MOV  A,M   7E
0538  LXI  H     21 04E6
053B  MOV  M,A   77
      * LINE   73
      * ENDIF

      * LOAD OUT OF SEQUENCE
0532  EQU              053C

      * LINE   74
      * CALL   EXTFUNC          TEMP1
      * PARM   THCHI          0
053C  CALL ADR   CD 0153
      * ENDPRM
      * :=     TEMP1          LCL
053F  LXI  H     21 04E4
0542  MOV  M,A   77
      * LINE   75
      * >      LCL 127 TEMP1
0543  LXI  H     21 04E4
0546  MOV  A,M   7E
0547  MVI  D     16 7F
0549  SUB  D     92
054A  JP   ADR   F2 0551
054D  XRA  A     AF
054E  JMP  ADR   C3 0553
0551  MVI  A     3E 01
      * IF     TEMP1
0553  ORA  A     B7
0554  JZ   ADR   CA AAAA
      * LINE   75
      * RETURN 0
0557  RETURN     C9
      * ENDIF

      * LOAD OUT OF SEQUENCE
0555  EQU    .         0558

      * LINE   76
      * IF     COMPL
0558  LXI  H     21 04E6
055B  MOV  A,M   7E
055C  ORA  A     B7
055D  JZ   ADR   CA AAAA
      * LINE   76
      * .C.    LCL          TEMP1
0560  LXI  H     21 04E4
0563  MOV  A,M   7E
0564  CMA        2F
      * +      TEMP1 1 TEMP2
0565  MVI  D     16 01
0567  ADD  D     82
      * :=     TEMP2          LCL
0568  LXI  H     21 04E4
056B  MOV  M,A   77
      * LINE   77
      * :=     FALSE          COMPL
056C  LXI  H     21 0407
056F  MOV  A,M   7E
0570  LXI  H     21 04E6
0573  MOV  M,A   77
      * LINE   78
      * ENDIF
```

```
      * LINE    79
      * ARYLOC VECTOR COL TEMP1
0574  MVI  B      C6 00
0576  LXI  H      21 04E7
0579  MOV  C,M    4E
057A  LHLD ADR    2A 04E9
057D  DAD  B      09
057E  PUSH H      E5
      * :=        LCL           TEMP1
057F  LXI  H      21 04E4
0582  MOV  A,M    7E
0583  POP  H      E1
0584  MOV  M,A    77
      * LINE    80
      * +         COL 1 TEMP1
0585  LXI  H      21 04E7
0588  MOV  A,M    7E
0589  MVI  D      16 01
058B  ADD  D      82
      * :=        TEMP1         COL
058C  LXI  H      21 04E7
058F  MOV  M,A    77
      * -         CNT 1 TEMP1
0590  LXI  H      21 04E8
0593  MOV  A,M    7E
0594  MVI  D      16 01
0596  SUB  D      92
      * :=        TEMP1         CNT
0597  LXI  H      21 04E8
059A  MOV  M,A    77
      * LINE    81
      * IF        CNT
059B  LXI  H      21 04E8
059E  MOV  A,M    7E
059F  ORA  A      B7
05A0  JZ   ADR    CA AAAA
      * LINE    81
      * CALL      EXTPROC          0
      * PARM      CHCUT         0
      * PARM      COMMA         0
05A3  LXI  H      21 0403
05A6  MOV  C,M    4E
05A7  PUSH B      C5
      * ENDPRM
05A8  POP  B      C1
05A9  MOV  A,C    79
05AA  CALL ADR    CD 00A6
      * ENDIF

      * LOAD OUT OF SEQUENCE
05A1  EQU               05AD

      * LINE    82
      * ENDWH

      * LOAD OUT OF SEQUENCE
050F  EQU               05B0

05AD  JMP  ADR    C3 0509
      * LINE    83
      * RETURN 0
05B0  RETURN      C9
```

```
      **** PROC PRINTVECTOR ****
      * LOCAL LCL
05B1  EQU             00
      * LOCAL CNT
05B2  EQU             00
      * VAL PARM PTR
05B3  EQU             00
      * REF PARM VECTCR
05B4  EQU             0000
      * SAVE RETURN ADDR
05B6  POP  D      D1
      * PUT VALUES IN PARMS
05B7  POP  B      C1
05B8  LXI  H      21 05B3
05BB  MOV  M,C    71
05BC  POP  H      E1
05BD  SHLD ADR    22 05B4
      * PUT RETURN ADDR BACK ON STACK
05C0  PUSH D      D5
      * LINE    87
      * :=        N             CNT
05C1  LXI  H      21 0405
05C4  MOV  A,M    7E
05C5  LXI  H      21 05B2
05C8  MOV  M,A    77
      * LINE    88
      * CALL      EXTPROC          0
      * PARM      CRLF          0
      * PARM      0             0
05C9  MVI  C      0E 00
05CB  PUSH B      C5
      * ENDPRM
05CC  POP  B      C1
05CD  MOV  A,C    79
05CE  CALL ADR    CD 0181
      * LINE    89
      * WHILE '0
      * WHTEST CNT
05D1  LXI  H      21 05B2
05D4  MOV  A,M    7E
05D5  ORA  A      B7
05D6  JZ   ADR    CA AAAA
      * LINE    90
      * ARYLOC VECTOR PTR TEMP1
05D9  MVI  B      06 00
05DB  LXI  H      21 05B3
05DE  MOV  C,M    4E
05DF  LHLD ADR    2A 05B4
05E2  DAD  B      09
05E3  PUSH H      E5
      * :=        TEMP1         LCL
05E4  POP  H      E1
05E5  MOV  A,M    7E
05E6  LXI  H      21 05B1
05E9  MOV  M,A    77
      * LINE    91
      * .A.       LCL 128 TEMP1
05EA  LXI  H      21 05B1
05ED  MOV  A,M    7E
05EE  MVI  D      16 80
05F0  ANA  D      A2
      * IF        TEMP1
```

```
        * CALL   EXTPRCC          0
        * PARM   CHOUT            0
        * PARM   MINUS            0
C5F5    LXI  H     21 0400
05F8    MOV  C,M   4E
05F9    PUSH B     C5
        * ENDPRM
05FA    POP  B     C1
05FB    MOV  A,C   79
05FC    CALL ADR   CD 00A6
        * LINE   93
        * .C.    LCL             TEMP1
05FF    LXI  H     21 0581
0602    MOV  A,M   7E
0603    CMA        2F
        * +      TEMP1 1 TEMP2
0604    MVI  D     16 01
0606    ADD  D     82
        * :=     TEMP2            LCL
0607    LXI  H     21 0581
060A    MOV  M,A   77
        * LINE   94
        * ELSEIF

        * LOAD OUT OF SEQUENCE
05F3    ECU              060E

060B    JMP  ADR   C3 AAAA
        * LINE   95
        * CALL   EXTPROC          0
        * PARM   CHOUT            0
        * PARM   BLANK            0
060E    LXI  H     21 0401
0611    MOV  C,M   4E
0612    PUSH B     C5
        * ENDPRM
0613    POP  B     C1
0614    MOV  A,C   79
0615    CALL ADR   CD 00A6
        * LINE   96
        * ENDIF

        * LOAD OUT OF SEQUENCE
060C    EQU              0618

        * LINE   97
        * CALL   EXTPRCC          0
        * PARM   THCHO            0
        * PARM   LCL          0
0618    LXI  H     21.0581
061B    MOV  C,M   4E
061C    PUSH B     C5
        * ENDPRM
061D    POP  B     C1
061E    MOV  A,C   79
061F    CALL ADR   CD C132
        * LINE   98
        * IF     CNT
0622    LXI  H     21 0582
0625    MOV  A,M   7E
0626    ORA  A     B7
0627    JZ   ADR   CA AAAA
        * LINE   98
```

```
C62A    LXI  H     21 0403
062D    MUV  C,M   4E
062E    PUSH B     C5
        * ENDPRM
062F    POP  B     C1
C630    MOV  A,C   79
0631    CALL ADR   CD 00A6
        * ENDIF

        * LOAD OUT OF SEQUENCE
C62E    EQU              0634

        * LINE   99
        * -        CNT 1 TEMP1
0634    LXI  H     21 0582
0637    MOV  A,M   7E
0638    MVI  D     16 01
063A    SUB  D     92
        * :=       TEMP1          CNT
063B    LXI  H     21 0582
063E    MOV  M,A   77
        * +        PTR 1 TEMP1
063F    LXI  H     21 0583
0642    MOV  A,M   7E
0643    MVI  D     16 01
0645    ADD  D     82
        * :=       TEMP1          PTR
0646    LXI  H     21 0583
0649    MOV  M,A   77
        * LINE   100
        * ENDWH

        * LOAD OUT OF SEQUENCE
0507    EQU              064D

064A    JMP  ADR   C3 0501
        * LINE   101
        * RETURN 0
064D    RETURN     C9
        * LINE   103
        * ENDPRC
        * PROC   PRINTMATRIX

        **** PROC PRINTMATRIX ****
        * LOCAL RCNT
064E    EQU              00
        * LOCAL PTR
064F    EQU              00
        * LINE   105
        * :=       N              RCNT
0650    LXI  H     21 0405
0653    MOV  A,M   7E
0654    LXI  H     21 064E
0657    MOV  M,A   77
        * :=       0              PTR
0658    MVI  A     3E 00
065A    LXI  H     21 064F
065D    MOV  M,A   77
        * LINE   106
        * WHILE  0
        * WHTEST RCNT
065E    LXI  H     21 064E
0661    MOV  A,M   7E
```

```
      * CALL    PRINTVECTOR         0
      * PARM    H        0
0666  LXI  H      21 0408
0669  PUSH H      E5
      * PARM    PTR          0
066A  LXI  H      21 064F
066D  MOV  C,M    4E
066E  PUSH B      C5
      * ENDPRM
066F  CALL ADR    CD 0586
      * LINE    108
      * -       RCNT 1 TEMP1
0672  LXI  H      21 064E
0675  MOV  A,M    7E
0676  MVI  D      16 01
0678  SUB  D      92
      * :=      TEMP1         RCNT
0679  LXI  H      21 064E
067C  MOV  M,A    77
      * LINE    109
      * +       PTR N TEMP1
067D  LXI  H      21 064F
0680  MOV  A,M    7E
0681  LXI  H      21 0405
0684  MOV  D,M    56
0685  ADD  D      82
      * :=      TEMP1         PTR
0686  LXI  H      21 064F
0689  MOV  M,A    77
      * LINE    110
      * ENDWH

      * LOAD OUT OF SEQUENCE
0664  EQU            068D

068A  JMP  ADR    C3 065E
      * LINE    111
      * RETURN 0
068D  RETURN      C9
      * LINE    113
      * ENDPRC
      * PROC    QUERY

      **** FUNC QUERY ****
      * LINE    115
      * CALL    PRINT          0
      * PARM    UK         0
068E  LXI  H      21 0460
0691  PUSH H      E5
      * PARM    14           0
0692  MVI  C      0E CE
0694  PUSH B      C5
      * ENDPRM
0695  CALL ADR    CD 04A1
      * LINE    116
      * CALL    EXTFUNC          TEMP1
      * PARM    CHIN           0
0698  CALL ADR    CD 0009
      * ENDPRM
      * =       TEMP1 NCHAR TEMP2
069B  LXI  H      21 0404
069E  MOV  D,M    56
069F  SUB  D      92

06A7  MVI  A      3E 01
      * IF      TEMP2
06A9  ORA  A      B7
06AA  JZ   ADR    CA AAAA
      * LINE    116
      * RETURN FALSE
06AD  LXI  H      21 0407
06B0  MOV  A,M    7E
06B1  RETURN      C9
      * LINE    117
      * ELSEIF

      * LOAD OUT OF SEQUENCE
06AB  EQU            06B5

06B2  JMP  ADR    C3 AAAA
      * LINE    117
      * RETURN TRUE
06B5  LXI  H      21 0406
06B8  MOV  A,M    7E
06B9  RETURN      C9
      * LINE    118
      * ENDIF

      * LOAD OUT OF SEQUENCE
06B3  EQU            06BA

      * LINE    120
      * ENDPRC
      * PROC    TRANSFORM

      **** PROC TRANSFORM ****
      * LOCAL COL
06BA  EQU            00
      * LOCAL PTR
06BB  EQU            00
      * LOCAL LCL
06BC  EQU            00
      * LOCAL CCNT
06BD  EQU            00
      * LOCAL RCW
06BE  EQU            00
      * LOCAL RCNT
06BF  EQU            00
      * REF PARM RESULTV
06C0  EQU            0000
      * REF PARM DATAV
06C2  EQU            0000
      * SAVE RETURN ADDR
06C4  POP  D      D1
      * PUT VALUES IN PARMS
06C5  POP  H      E1
06C6  SHLD ADR    22 06C0
06C9  POP  H      E1
06CA  SHLD ADR    22 06C2
      * PUT RETURN ADDR BACK ON STACK
06CD  PUSH D      D5
      * LINE    122
      * :=      N             RCNT
06CE  LXI  H      21 0405
06D1  MOV  A,M    7E
06D2  LXI  H      21 06BF
06D5  MOV  M,A    77
```

```
[illegible] LXI  H      21 [illegible]
0608  MOV  M,A     77
      * :=         0          PTR
060C  MVI  A       3E 00
060E  LXI  H       21 06BB
06E1  MOV  M,A     77
      * LINE  123
      * WHILE  0
      * WHTEST RCNT
06E2  LXI  H       21 06BF
06E5  MOV  A,M     7E
06E6  ORA  A       B7
06E7  JZ   ADR     CA AAAA
      * LINE   124
      * :=         0          LCL
06EA  MVI  A       3E 00
06EC  LXI  H       21 06BC
06EF  MOV  M,A     77
      * :=         N          CCNT
06F0  LXI  H       21 0405
06F3  MOV  A,M     7E
06F4  LXI  H       21 06BD
06F7  MOV  M,A     77
      * :=         0          COL
06F8  MVI  A       3E 00
06FA  LXI  H       21 06BA
06FD  MOV  M,A     77
      * LINE   125
      * WHILE  C
      * WHTEST CCNT
06FE  LXI  H       21 06BD
0701  MOV  A,M     7E
0702  ORA  A       B7
0703  JZ   ADR     CA AAAA
      * LINE   126
      * ARYLOC H PTR TEMP1
0706  MVI  B       06 00
0708  LXI  H       21 06BB
070B  MOV  C,M     4E
070C  LXI  H       21 0408
070F  DAD  B       09
0710  MOV  A,M     7E
      * =          TEMP1 262142 TEMP2
0711  MVI  D       16 FF
0713  SUB  D       92
0714  JZ   ADR     CA 071B
0717  XRA  A       AF
0718  JMP  ADR     C3 071D
071B  MVI  A       3E 01
      * IF         TEMP2
071D  ORA  A       B7
071E  JZ   ADR     CA AAAA
      * LINE   127
      * ARYLOC DATAV COL TEMP1
0721  MVI  B       06 00
0723  LXI  H       21 06BA
0726  MOV  C,M     4E
0727  LHLD ADR     2A 06C2
072A  DAD  B       09
072B  MOV  C,M     4E
072C  PUSH B       C5
      * -          LCL TEMP1 TEMP2
072D  LXI  H       21 06BC
[illegible] MOV A,M 7E
```

```
      * :=         TEMP2          LCL
0734  LXI  H       21 06BC
0737  MOV  M,A     77
      * LINE   128
      * ELSEIF

      * LOAD OUT OF SEQUENCE
071F  EQU                0738

0738  JMP  ADR     C3 AAAA
      * LINE   129
      * ARYLOC DATAV COL TEMP1
073B  MVI  B       06 00
073D  LXI  H       21 06BA
0740  MOV  C,M     4E
0741  LHLD ADR     2A 06C2
0744  DAD  B       09
0745  MOV  C,M     4E
0746  PUSH B       C5
      * +          LCL TEMP1 TEMP2
0747  LXI  H       21 06BC
074A  MOV  A,M     7E
074B  POP  B       C1
074C  MOV  D,C     51
074D  ADD  D       82
      * :=         TEMP2          LCL
074E  LXI  H       21 06BC
0751  MOV  M,A     77
      * LINE   130
      * ENDIF

      * LOAD OUT OF SEQUENCE
0739  EQU                0752

      * LINE   131
      * -          CCNT 1 TEMP1
0752  LXI  H       21 06BD
0755  MOV  A,M     7E
0756  MVI  D       16 01
0758  SUB  D       92
      * :=         TEMP1          CCNT
0759  LXI  H       21 06BD
075C  MOV  M,A     77
      * LINE   132
      * +          PTR 1 TEMP1
075D  LXI  H       21 06BB
0760  MOV  A,M     7E
0761  MVI  D       16 01
0763  ADD  D       82
      * :=         TEMP1          PTR
0764  LXI  H       21 06BB
0767  MOV  M,A     77
      * +          COL 1 TEMP1
0768  LXI  H       21 06BA
076B  MOV  A,M     7E
076C  MVI  D       16 01
076E  ADD  D       82
      * :=         TEMP1          COL
076F  LXI  H       21 06BA
0772  MOV  M,A     77
      * LINE   133
      * ENDWH
```

```
0773  JMP  ADR    C3 06FE
      * LINE   134
      * ARYLOC RESULTV ROW TEMP1
0776  MVI  B      06 00
0778  LXI  H      21 06BE
077B  MOV  C,M    4E
077C  LHLD ADR    2A 06C0
077F  DAD  B      09
0780  PUSH H      E5
      * :=    LCL       TEMP1
0781  LXI  H      21 06BC
0784  MOV  A,M    7E
0785  POP  H      E1
0786  MOV  M,A    77
      * LINE   135
      * -     RCNT 1 TEMP1
0787  LXI  H      21 06BF
078A  MOV  A,M    7E
078B  MVI  D      16 01
078D  SUB  D      92
      * :=    TEMP1        RCNT
078E  LXI  H      21 06BF
0791  MOV  M,A    77
      * +     ROW 1 TEMP1
0792  LXI  H      21 06BE
0795  MOV  A,M    7E
0796  MVI  D      16 01
0798  ADD  D      82
      * :=    TEMP1        ROW
0799  LXI  H      21 06BE
079C  MOV  M,A    77
      * LINE   136
      * ENDWH

      * LOAD OUT OF SEQUENCE
06E8  EQU             07A0

079D  JMP  ADR    C3 06E2
      * LINE   137
      * RETURN 0
07A0  RETURN      C9
      * LINE   139
      * ENDPRC
      * PROC    HADAMARD

      **** PROC HADAMARD ****
      * LINE    141
      * CALL    PRINT          0
      * PARM    HEADER         0
07A1  LXI  H      21 0448
07A4  PUSH H      E5
      * PARM    24             0
07A5  MVI  C      0E 18
07A7  PUSH B      C5
      * ENDPRM
07A8  CALL ADR    CD 04A1
      * LINE    142
      * CALL    PRINTMATRIX
      * ENDPRM
07AB  CALL ADR    CD 0650
      * LINE    143
      * WHILE   0
      * WHTEST  1
```

```
      * LINE    144
      * CALL    INVECTOR       0
      * PARM    DATA           0
07B4  LXI  H      21 048D
07B7  PUSH H      E5
      * ENDPRM
07B8  CALL ADR    CD 04EB
      * LINE    145
      * CALL    PRINTVECTOR    0
      * PARM    DATA           0
07BB  LXI  H      21 048D
07BE  PUSH H      E5
      * PARM    0              0
07BF  MVI  C      0E 00
07C1  PUSH B      C5
      * ENDPRM
07C2  CALL ADR    CD 05B6
      * LINE    146
      * CALL    QUERY          TEMP1
      * ENDPRM
07C5  CALL ADR    CD 068E
      * IF      TEMP1
07C8  ORA  A      B7
07C9  JZ   ADR    CA AAAA
      * LINE    147
      * CALL    TRANSFORM      0
      * PARM    DATA           0
07CC  LXI  H      21 048D
07CF  PUSH H      E5
      * PARM    RESULT         0
07D0  LXI  H      21 0495
07D3  PUSH H      E5
      * ENDPRM
07D4  CALL ADR    CD 06C4
      * LINE    148
      * CALL    PRINT          0
      * PARM    OUTMESS        0
07D7  LXI  H      21 0480
07DA  PUSH H      E5
      * PARM    12             0
07DB  MVI  C      0E 0C
07DD  PUSH B      C5
      * ENDPRM
07DE  CALL ADR    CD 04A1
      * LINE    149
      * CALL    PRINTVECTOR    0
      * PARM    RESULT         0
07E1  LXI  H      21 0495
07E4  PUSH H      E5
      * PARM    0              0
07E5  MVI  C      0E 00
07E7  PUSH B      C5
      * ENDPRM
07E8  CALL ADR    CD 05B6
      * LINE    150
      * ENDIF

      * LOAD OUT OF SEQUENCE
07CA  EQU             07EB
      * LINE    151
      * ENDWH
```

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC


```
07EB  JMP  ADR    C3 07AE
      * LINE    152
      * ENDPRC
      * START  HADAMARD
07EE  RST  1      CF

 MAXIMUM ADDRESS IS 07EE
 START EXECUTION AT ADDRESS 07A1
```

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

UVA LDR


```
;16040020D20302C4E0801000101010101010101FF01FF01FF081E
;16041601FF0101FFFF0101FFFF01FFFF0101FFFF0101010101010D30
;16042CFFFFFFFF01FF01FFFF01FF010101FFFFFFFF010101FF113E
;160442FF01FF0101FF0D0A0A0A0A48414441440415244204D410A6D
;16045B5452495B0D0A0A0A0D0A0A4F4B3F20284E3D4E4F29200893
;16046E0D0A0A494E505554205645435944F520D0A0A0D0A0A52088C
;160484455354C54C0D0A0A0000000000000000000000000000648
;17049A000000000C0000C01C1219E0471E1229F04D53E002190040AF2
;18048177219D047E219E045692CAC0043E0187CAAAAA0600219D040D95
;1604C94E2A9F04094EC5C179CDA6002190047E160182219D040C5E
;0104DF770557
;0204C2E30409A7
;17049E0C3B204C900000000000000001E122E904D5216E04E50E120C67
;1604F7C5CDA1042105047E21E804773E0021E7047721E8047E0C8C
;18050DB7CAAAAA3E0021E5047721070047E21E60477CDD9002100040D80
;160525569Z2CA2E05AFC330053E0187CAAAAA2106047E21E6040D8F
;01053B770593
;0205323C050A70
;17053CCD530121E4047721E4047E167F92E25105AFC353053E010DF3
;C505538 7CAAAAAC908F6
;0205555805CAAF
;17055821E6047EB7CAAAAA21E4047E2F16018221E4047721070040DC8
;05056F7E21E604770774
;02055E74050AD4
;170574060021E7044E2AE90409E521E4047EE17721E7047E16010D70
;180568B6221E7047721E8047E16019221E8047721E8047EB7CAAAAA0FC0
;0A05A3210304 4EC5C179CDA6000995
;0205A1AD050850
;02050FB0050AC1
;1705ADC30905C900000000000D1C121B30571E122840505210504 0CF5
;1705C47E21B205770E00C5C179CD810121B2057EB7CAAAAA06C00F35
;16050821B3054E2AB40509E5E17E21B105772181057E1680A20E23
;1605F1B7CAAAAA2100044EC5C179CDA60021B1057E2F1601820EDE
;0406072181057 70759
;0205F30E060C03
;0D0608C3AAAA2101044EC5C179CDA6000C15
;02060C18060C26
;1606182 B1054EC5C179CD32012182057EB7CAAAAA2103044E0EF3
;06062EC5C179CDA60009A6
;02062834C60C5E
;16063421B2057E16019221B2057721B3057E16018221B305770CD8
;02C5D74DC60C26
;17064AC3D105C900C02105047E214E06773E00214F0677214E060BF7
;1706617EB7CAAAAA210804E5214F064EC5CD8605214E067E16010EF8
;12067892214E06772 14F067E21050456822 14F06770AEB
;0206648D060CF3
;16068AC35E06C9216004E50E0EC5CDA104CDD90021040456920F04
;1206A0CAA706AFC3A9063E0187CAAAAA21C7047EC90ED1
;0206AB35060D62
;060682C3AAAA21C6047EC90A43
;020683BA060D6F
;17068A0000000000000000000000C1E122C0C6E122C2C6D52105040C35
;16060 17E218F06773E00218E06773E00218B067721 8F067E870E0E
;1606E7CAAAAA3E00218C0677210504 7E21BD06773E00218A0600D5
;1606FD77218D067EE7CAAAAA0600218B064E21080 4097E16FF0EC0
;17071392CA1B07AFC31D073E01B7CAAAAA060021BA064E2AC2060F79
;0E072A094EC5218C067EC15192218C06770CB3
;02071F3B070E5C
;16075221BD067E16019221BD067721BB067E16018221BB06770E20
;0807 6821BA067E16018221BA06770AC3
;02070476070E7C
;17C773C3FE0606 00218E064F2AC00609E5218C067EE177218F061007
;13078A7E16019221BF067721BE067E16018221BE06770D79
;0206E8A0070E8A
;170790C3E206C9214804E50E18C5CDA104CD50063E01B7CAAAAA120E
;18078421 8D04E5CDEB04218D04E50E00C5CD8605CD8E06B7CAAAAA1347
;1607CC218D04E5219504E5CDC406218D04E50E0CC5CDA104219504 1146
;0707E4E50E00C5CD86050B28
;0207CAE8070F87
;0207B2EE070FA2
;0407EBC3AE07CFCA36
;00
```

```
.J Ø791

HADAMARD MATRIX

 Ø1, Ø1, Ø1, Ø1, Ø1, Ø1, Ø1, Ø1
 Ø1,-Ø1, Ø1,-Ø1, Ø1,-Ø1, Ø1,-Ø1
 Ø1, Ø1,-Ø1,-Ø1, Ø1, Ø1,-Ø1,-Ø1
 Ø1,-Ø1,-Ø1, Ø1, Ø1,-Ø1,-Ø1, Ø1
 Ø1, Ø1, Ø1, Ø1,-Ø1,-Ø1,-Ø1,-Ø1
 Ø1,-Ø1, Ø1,-Ø1,-Ø1, Ø1,-Ø1, Ø1
 Ø1, Ø1,-Ø1,-Ø1,-Ø1,-Ø1, Ø1, Ø1
 Ø1,-Ø1,-Ø1, Ø1,-Ø1, Ø1, Ø1,-Ø1
INPUT VECTOR
-Ø1,-Ø2,-Ø3,-Ø4, Ø1, Ø2, Ø3, Ø4
-Ø1,-Ø2,-Ø3,-Ø4, Ø1, Ø2, Ø3, Ø4
OK? (N=NO)Y
RESULT

 ØØ, ØØ, ØØ, ØØ,-14, Ø4, Ø8, ØØ
INPUT VECTOR
 ØØ, ØØ, ØØ, ØØ,-14, Ø4, Ø8, ØØ
 ØØ, ØØ, ØØ, ØØ,-14, Ø4, Ø8, ØØ
OK? (N=NO)Y
RESULT

-Ø8,-1Ø,-18,-2Ø, Ø8, 1Ø, 18, 2Ø
INPUT VECTOR
```

Note: This page is a typed duplicate of the teletype print-out produced by HADAM executing on the UVA Modular System Intel 8080.

Appendix II. SIMPL Quads Used by SIMPL-M

ID	Name	Quad Fields and Types*		
Ø	LINE	A = I		
1	=	A = S,I,T	B = S,I,T	R=T
2	<>	"	"	"
3	<	"	"	"
4	<=	"	"	"
5	>	"	"	"
6	>=	"	"	"
7	+	"	"	"
8	-	"	"	"
9	*	"	"	"
10	/	"	"	"
11	.A. (bit and)	"	"	"
12	.V. (bit or)	"	"	"
13	.X. (bit xor)	"	"	"
14	.RL. (rt log shift)	"	"	"
15	.LL. (lf log shift)	"	"	"
16	NEGATE	A = S,I,T		"
17	.C. (bit compl)	"		"
18	.NOT. (logical)	"		"
19	.AND. (logical)	A = S,I,T	B = S,I,T	R = T
20	.OR. (logical)	"	"	"
30	:= (becomes)	A = S,I,T		R = S
31	IF	A = S,I,T		

ID	Name	Quad Fields and Types*		
32	ELSEIF			
33	ENDIF			
34	WHILE	A = I		
35	WHTEST	A = S,I,T		
36	ENDWH			
37	INITCS			
38	CSTEST	A = S,I,T		R = I
39	CASE	A = I		
40	CSEND			
41	CSELSE			
42	ENDCS			
43	CALL	A = S		R = I,T
44	PARM	A = S,I,T		R = I
45	ENDPRM			
46	ARYLOC	A = S	B = S	R = T
47	START	A = S,I		
48	RETURN	A = S		
49	EXIT	A = I		
50	PROC	A = S		
51	ENDPRC			

* T=Temporary, I=Immediate, S=SYMTAB Ptr.

Appendix III. <u>Typical Quad Sequences</u>

Note: All LINE quads have been omitted.

IF Statement:

```
IF DOG = CAT THEN
   FRUIT := APPLE
ELSE
   FRUIT := Ø
END
```

Quads Produced:

ID	A	B	R
=	DOG	CAT	TEMP1
IF	TEMP1		
:=	APPLE		FRUIT
ELSEIF			
:=	Ø		FRUIT
ENDIF			

WHILE Statement:

```
CNT := 1Ø
WHILE CNT DO
   IF CNT = OFF THEN EXIT END
   CNT := CNT - 1
END/*WHILE*/
```

Quads Produced:

ID	A	B	R
:=	1Ø		CNT
WHILE	Ø		
WHTEST	CNT		
=	CNT	OFF	TEMP1
IF	TEMP1		
EXIT			
-	CNT	1	TEMP2
:=	TEMP2		CNT
ENDWH			

CASE Statement:

```
CASE INDEX-1 OF
   ≥11≥ ≥12≥  INDEX := 1Ø
   ≥3≥        INDEX := Ø
ELSE INDEX := 8
END/*CASE*/
```

Quads Produced:

ID	A	B	R
INITCS			
-	INDEX	1	TEMP1
CSTEST	TEMP1		
CASE	11		
CASE	12		
:=	1Ø		INDEX
CSEND			
CASE	3		
:=	Ø		INDEX
CSEND			
CSELSE			
:=	8		INDEX
CSEND			
ENDCS			

PROC Statement:

```
PROC ADDIT(INT APPLE, INT WHICH, REF INT DOG)
```

Quads Produced:

ID	A	B	R
PROC	ADDIT		

Note: Parameters must be sought in the parameter/local chain as described in Section 2.2.3.

CALL Statement:

CALL ADDIT(FRUIT,LEAST(BOTH),ANIMAL)

Note: LEAST is a FUNC

Quads Produced:

ID	A	B	R
CALL	ADDIT		
PARM	FRUIT		Ø
CALL	LEAST		TEMP1
PARM	BOTH		Ø
ENDPRM			
PARM	TEMP1		Ø
PARM	ANIMAL		1
ENDPRM			

Array Reference: All array references produce an ARYLOC quad which assigns the array location or value to a temporary.

PILE(12) := HEAP(Ø) + 5

Quads Produced:

ID	A	B	R
ARYLOC	HEAP	Ø	TEMP1
+	TEMP1	5	TEMP2
ARYLOC	PILE	12	TEMP3
:=	TEMP2		TEMP3

The following quads need further explanation:

ID	Name	Explanation
Ø	LINE	A is the line number
34	WHILE	A is the designator and is ignored
38	CSTEST	R is data type and is ignored
39	CASE	A is the case number
43	CALL	A is the PROC/FUNC and R = temporary for FUNC and Ø for PROC
44	PARM	A is the argument and R = 1 for REF and Ø for value
47	START	A is the main PROC or Ø if none
48	RETURN	A is SYMTAB address of value to return or Ø if none
49	EXIT	A is the designator and is ignored
50	PROC	A is the PROC/FUNC

Appendix IV. <u>Procedure for Re-Compiling SIMPL-M</u>

Step 1. Compile a new version of CODGEN

```
Job Card
UPDATE,N.
REWIND,COMPILE.
ATTACH,SIMPLT.
REQUEST,COMPAS,*PF.
SIMPLT,S,COMPILE.
CATALOG,COMPAS,CODGEN,RP=Ø.
7/8/9
*DECK CODGEN
Source Deck
6/7/8/9
```

Step 2. Assemble output code in file CODGEN, merge new code generator with existing SIMPLT package, merge this with overlay structure, and save this as the new SIMPLM compiler.

```
Job Card
ATTACH,COMPAS,CODGEN.
COMPASS,L=Ø,I=COMPAS.
ATTACH,LCL,SEPFLS.
COPYL,LCL,LGO,NEW.
ATTACH,OVER.
COPYL,OVER,NEW,OVFLS.
REQUEST,ABS,*PF.
ATTACH,RSL.
LIBRARY,RSL.
RFL,75ØØØ.
LOAD,OVFLS.
NOGO,ABS.
CATALOG,ABS,SIMPLM.
6/7/8/9
```

Step 3. The SIMPL-M compiler can now be utilized with the procedure outlined in reference 2.

Appendix V. SIMPL-T on the UVA CDC Cyber 172

SIMPL-T as it is available on the Cyber 172 is cumbersome in that it produces assembly code which must be assembled before it can be executed. See reference 9 for a complete explanation of CDC SIMPL-T.

At present SIMPL-T is only available on tape. When SIMPL-T becomes available as a disk file, Step 1 will not be necessary.

Step 1. Read SIMPL-T and its library from tape.

```
Operator request card
Job Card - specify MT1
REQUEST,OLDPL,VSN=SIMPLT,HY,NORING.
REWIND,OLDPL.
REQUEST,SIMPLT,*PF.
COPYBF,OLDPL,SIMPLT.
CATALOG,SIMPLT,RP=30.
REQUEST,RSL,*PF.
COPYBF,OLDPL,RSL.
CATALOG,RSL,RP=3Ø.
6/7/8/9
```

Step 2. Compile SIMPL-T program using UPDATE.

```
Job Card
UPDATE,N.
REWIND,COMPILE.
ATTACH,SIMPLT.
ATTACH,RSL.
LIBRARY,RSL.
REQUEST,COMPAS,*PF.
SIMPLT,S,COMPILE.
CATALOG,COMPAS,RP=Ø.
7/8/9
*DECK program name
Source deck
6/7/8/9
```

Step 3. Once compilation errors have been eliminated, assemble COMPAS.

```
Job Card
ATTACH,COMPAS.
COMPASS,L=Ø,I=COMPAS.
LGO.
6/7/8/9
```

Appendix VI. SIMPL-M User's Manual

SIMPL-M: A Structured Programming Language for Microcomputers

A User's Manual

by

James B. Bladen

Victor R. Basili

Albert J. Turner

Prepared for the
Computer Science Department of the UNIVERSITY OF VIRGINIA
ENGINEERING SCHOOL, Charlottesville, Va. 22901.

PREFACE

SIMPL-M is a member of the SIMPL family of languages as designed by Victor R. Basili and Albert J. Turner of the University of Maryland. This user's manual is based on their manual (1).

The SIMPL-M compiler is implemented on the UVA CDC Cyber 172 computer. The original CDC implementation of SIMPL was written by John G. Perry, Jr. of Dahlgren Labs, Va. Mr. Perry's system is an implementation of SIMPL-T and is documented in (2).

The SIMPL-M compiler runs on the CDC and generates machine code for the Intel 8080 microcomputer. Reference (3) gives a thorough explanation of the Intel 8080 assembly and machine codes.

The loader format generated by SIMPL-M conforms to the requirements of the UVA Modular System (4). The paper referenced in (4) is reproduced in Appendix IV.

Internal documentation of the SIMPL-M code generator is contained in <u>SIMPL-M: A Cross-Compiler for the Intel 8080 Microcomputer</u>, a master's thesis by James B. Bladen (5).

CONTENTS

1. Introduction

SIMPL-M is a cross-compiler that executes on the CDC 6000 and Cyber series computers, and generates Intel 8080 machine code. SIMPL-M also produces an assembly language style printout which lists the quads generated by the compiler and the Intel code produced for each quad. The current implementation produces a paper tape which is read into the microcomputer via a teletype tape reader (See Appendix I for a guide for executing SIMPL-M).

SIMPL-M is the SIMPL-T compiler with a new code generator module substituted for the CDC6000 module. Since the Intel 8080 microcomputer has greatly reduced capability and size compared to the CDC, many of the features offered in CDC SIMPL-T have been eliminated. Some new features have been added due to the flexible nature of microcomputers.

1.1 Features of SIMPL-T not Supported by SIMPL-M.

1. Recursive procedure calls.
2. String data and Partwords.
3. Files.
4. WHILE descriptors.
5. Disk-resident program library.

1.2 New Features Offered by SIMPL-M.

1. Start load specification. Any memory address can be declared as a starting point for the program load. The default is address zero.
2. Address specification for external subroutines. Pre-programmed and loaded procedures can be executed by specifying the address in the EXTFUNC or EXTPROC procedure call. This takes advantage of pre-programmed modules often resident in the PROM of microcomputers..

1.3 SIMPL-M Restrictions.

1. The only data type is integer.
2. Integers must be in the range -128 to 127 decimal.
3. The only data structure is the one-dimensional array.
4. The maximum array length is 127.

These restrictions apply only to the present version of SIMPL-M. Future revisions will modify them.

2. The Basic SIMPL-M Language

2.1 Program Structure

The syntax for a SIMPL-M program is illustrated by

{<declaration list>} <segment list> START <identifier>

The <declaration list> defines the variables that may be used anywhere in the program. The <segment list> is a collection of procedures (subroutines) and functions, and <identifier> names the procedure with which execution is to begin. (The <segment list> may consist of only a single procedure.)

The following example illustrates this program structure.

```
INT X,Y,Z                                   } declaration list

PROC SUM(INT A, INT B)
   Z:=A+B
PROC MAINPROG                                 segment list
   X := 3
   Y := 4
   CALL SUM(X,Y)
START MAINPROG
```

Thus a SIMPL-M program contains a (possibly empty) set of global declarations and a set of procedures and functions. Execution begins with one of the procedures, and the procedures and functions are called as needed during execution.

2.1.1 Declarations

The initial declaration list of a program contains declarations for all variable identifier names that are global, A global identifier is an identifier that is known to all segments of a program.

2.1.1.1 Integer Declaration

An integer variable may have any integer value between -128 and 127, inclusive. An integer variable declaration consists of the keyword INT followed by one or more identifier names, separated by commas. Initialization may also be specified as illustrated by the following valid declaration

list.

```
INT X
INT CAT, DOG1
INT M=3, N=-1, I
```

In the above example M and N are initialized to the values 3 and -1, respectively. This means that these variables will have the specified values when execution of the program begins. The value of an uninitialized variable is initially zero.

2.1.1.2 Integer Array Declaration

The only data structure in SIMPL-M is the one-dimensional array. This is an ordered collection of elements, all of the same data type. The elements are numbered 0, 1, ..., n-1 , where n is the number of elements in the array.

Integer array declarations begin with the keywords INT ARRAY , and are completed by listing the array identifiers and the number of elements for each array. The number of elements must be a positive integer, and is enclosed in parentheses. For example,

```
INT ARRAY  TOTALS(10)
```

declares an array of 10 elements: TOTALS(0), TOTALS(1),..., TOTALS(9).

An array can also be initialized by specifying a list of values for the array elements. Initialization begins with the first element (number 0) and proceeds until the list is exhausted (or all array elements are exhausted). A repetition factor can be specified by enclosing the factor in parentheses following the initialization value.

Some examples are

```
INT ARRAY A(3), BAT(95), VECTOR(20)
INT ARRAY A1(10), B(5) = (2,3,-1)
INT ARRAY C(11) = (0,1,3(9))
```

The second declaration specifies that B(0) , B(1) , and B(2) are to be initialized to 2 , 3 , and -1 , respectively. The third declaration initializes C(0) to 0 , C(1) to 1 , and C(2)-C(10) to 3. See Section 1.3 for restrictions on SIMPL-M arrays.

2.1.1.3 Declaration List

A declaration list, such as the list of global declarations at the beginning of a program, consists of one or more declarations. Declarations follow one another with no separator (except blanks). More than one declaration for the same type can appear in a declaration list. All identifiers used in a program must be declared.

An example of a declaration list is

```
INT X, Y
INT I
INT ARRAY INPUTS(20),OUTPUTS(20)
INT SUM
```

2.1.2 Segments

A segment is a procedure or function definition. Segments contain a list of statements to be executed when the segment is invoked.

2.1.2.1 Procedures

The syntax for a procedure definition is illustrated by

```
PROC <identifier> {(<parameter list>)} {<local declaration list>}
     <statement list> {RETURN}
```

where <identifier> is the name of the procedure.

An example of a procedure definition is

```
PROC TEST (INT X, INT Y)
/* THIS PROC SUMS X AND Y */
INT Z
Z := X+Y
```

A procedure is a subroutine that, when invoked, executes its <statement list> and returns to the caller. A procedure may access any global identifier (unless the procedure has a local identifier by the same name) as well as its local identifiers and parameters.

The items of the <parameter list> , separated by commas, are of the form INT <identifier> or INT ARRAY <identifier> . These parameters are passed to the procedure when it is invoked (called)..

Integer parameters are passed by value (unless otherwise specified as in 2.3.7. This means that if a procedure changes the value of an integer parameter, the new value is effective only to that procedure. For example, if procedure P is defined by:

```
        PROC P(INT X)
          INT Y
          X := 7
```

and the statements

```
          X := 3
          CALL P(X)
          Y := X
```

are executed, then Y will become 3 (not 7).

Array parameters, however, are passed by reference. Logically, this means that the array itself is passed (rather than the value as for integer parameters). Thus any modification to an array parameter by a procedure will be a modification to the actual array passed as an argument by the caller.

2.1.2.2 Functions

The function definition syntax is illustrated by

```
INT FUNC <identifier> {(<parameter list>)} {<local declaration list>}
    {<statement list>} RETURN(<expression>)
```

A function is similar to a procedure. The main differences are

1) the value of <expression> is returned (as the value of the function evaluation) to be used in the same manner as the value of a variable would be used;

2) functions may not have side effects, that is, they may not change the values of any nonlocal variables or arrays.

2.1.2.3 Local Declarations

All local variables must be declared in the local declaration list. Local declarations are similar to global declarations, but initialization is not allowed. (The values of local variables at first entry to a segment are zero.)

2.1.3 Scope of Identifiers

Global identifiers, including segment names, are accessible from all segments unless a segment declares a local with the same name as a global. Local declarations override global declarations so that a global indentifier is not available to a segment in which that identifier is declared local.

Local identifiers are only accessible to the segment in which they are declared. Both globals and locals may be passed as parameters. The

value of all locals is undefined at entry to the segment, and locals do not necessarily retain their values between successive calls to the segment.

2.2 Comments and Blanks

Blanks may appear anywhere in a SIMPL-M program except within an identifier, symbol, keyword, or constant. Blanks are significant delimiters and may be needed as separators for identifiers or constants. For example,

```
IF X
```

and

```
IFX
```

are not equivalent.

A comment is any character string enclosed by /* and */ . A comment may appear anywhere that a blank may occur and has no effect on the execution of a program. The following illustrates a comment:

```
/* THIS IS A COMMENT. */
```

2.3 Statements

The syntactic entity <statement list> denotes any sequence of SIMPL-T statements. No separators (other than blanks) are used between statements.

2.3.1 Assignment Statement

The syntax of the assignment statement is given by

```
<variable> := <expression>
```

where <variable> is either a simple variable (i.e., an integer identifier) or a subscripted variable. The assignment statement causes the value of the <expression> to be assigned to the <variable> . Examples of valid SIMPL-T assignment statements are

```
X := Y+Z
X := Y=Z
A(I):= A(I+1)+A(J-2)*X
```

2.3.2 If Statement

The IF statement causes conditional execution of a sequence of one or

more statements. The syntax is

IF <expression>
THEN <statement list>$_1$
{ELSE <statement list>$_2$} END

At execution, the value of the <expression> determines the action taken. If the value is nonzero, <statement list>$_1$ is executed and <statement list>$_2$ (if there is an else part) is skipped. If the value is zero, <statement list>$_2$ (if it exists) is executed and <statement list>$_1$ is not executed. Execution proceeds with the next statement (following END) after execution of either <statement list> .

<u>Example</u>

```
IF X<3 .AND. Y<X
    THEN
      Y:=X
    ELSE
      X:=X+1
      Y:=Y-1
      IF X>Y
          THEN
             X:=Y
          END
    END
```

Note that the ELSE part of the main IF statement also contains an IF statement that will be executed only if the ELSE part is executed.

<u>Example</u>.

```
IF X THEN Y:=Y/X ELSE Y:=Y/2  END
```

This statement divides Y by X if X is nonzero and divides by 2 if X is zero.

2.3.3 <u>While Statement</u>

The WHILE statement provides a means of iteration (looping):

WHILE <expression> DO <statement list> END

The value of the <expression> determines the action at execution time, just

as for the IF statement. If the value of <expression> is nonzero, then <statement list> is executed; otherwise <statement list> is skipped and execution proceeds with the statement following END . However, if <statement list> is executed, then execution proceeds with the WHILE statement again. Thus if <expression> is nonzero, then <statement list> is executed until <expression> becomes zero.

Example. The following statement list sums the odd and even integers from 1 to 100.

```
ODD := 0
EVEN := 0
I := 0
WHILE I<100
DO
    I := I+1
    IF I/2 * 2 = I
        THEN /* EVEN INTEGER */
            EVEN := EVEN + I
        ELSE /* ODD */
            ODD := ODD + I
        END
END
```

2.3.3.1 Exit Statement

The EXIT statement provides a means of escaping from a WHILE loop. In its basic form, the statement

```
EXIT
```

causes the immediate termination of the (innermost) WHILE statement containing the EXIT statement. Execution proceeds as if the WHILE statement has terminated normally.

2.3.4 Case Statement

Exactly one of a group of statement lists may be executed by using the CASE statement. The syntax is illustrated by

```
CASE <expression> OF
  >n1>  <statement list>1
  >n2>  <statement list>2
          .
          .
          .
  >nk>  <statement list>k
  {ELSE <statement list>k+1} END
```

where each n_1, $n_2, \ldots, n_k$ is a constant or a negated constant.

If the value of <expression> is n_j, then <statement list>$_j$ is executed and the other statement lists are not executed. If <expression> does not evaluate to any of the n_i's, then the ELSE part (<statement list>$_{k+1}$) is executed, if there is an ELSE part, and none of the statement lists is executed if there is no ELSE part. The cases may be in any order, and more than one case designator n_i may be used with the same statement list, as is illustrated in the following example.

Example

```
CASE X*Y+Z OF
  >1>
       X := 3
  >-8> /* CASES NEED NOT BE IN ORDER*/
       IF X<Y
          THEN
             X := Y
          END
       Y := Y+L
  >4>6> /* CASES 4 AND 6 COINCIDE */
       X := 2
       Y := 3
  ELSE
       X := 0
  END
```

2.3.5 Call Statement

The CALL statement

CALL <identifier> {(<argument list>)}

causes the procedure named <identifier> to be executed. Each argument in the argument list may be an expression or an array, and the arguments must agree in number and type with the parameters in the procedure definition for the procedure that is called. Arguments in <argument list> are separated by commas.

Upon completion of the execution of the procedure, execution resumes with the statement following the CALL statement,

Example. To invoke the procedure DOIT with arguments X+Y and the array A , the statement

```
CALL DOIT (X+Y, A)
```

is used.

2.3.6 Example

```
PROC SORT (INT N, INT ARRAY A)
/* THIS PROCEDURE USES A BUBBLE SORT ALGORITHM TO SORT THE
   ELEMENTS OF ARRAY 'A' INTO ASCENDING ORDER. THE VALUE OF
   THE PARAMETER 'N' IS THE NUMBER OF ITEMS TO BE SORTED.  */
INT SORTED, /* SWITCH TO INDICATE WHETHER FINISHED */
   LAST,     /* LAST ELEMENT THAT NEED TO BE CHECKED */
   I,        /* FOR GOING THROUGH ARRAY */
   SAVE      /* FOR HOLDING VALUES TEMPORARILY */
IF N>1
  THEN /* SORT NEEDED */
   SORTED := 0 /* INDICATE NOT FINISHED */
   LAST := N-1 /* START WITH WHOLE ARRAY */

   WHILE .NOT. SORTED
     DO /* CHECK CURRENT SEQUENCE FOR CORRECTNESS */
       SORTED := 1 /* ASSUME FINISHED */
       I := 1      /* INITIALIZE ELEMENT POINTER */

       WHILE I <= LAST
         DO /* COMPARE ADJACENT ELEMENTS UP TO 'LAST' */
```

```
      IF A(I-1) > A(I)
        THEN /* OUT OF ORDER */
          SAVE := A(I)      /* INTERCHANGE */
          A(I) := A(I-1)    /* A(I) AND    */
          A(I-1) := SAVE    /* A(I-1)      */
          SORTED := 0       /* MAY NOT BE FINISHED */
        END
      I := I+1
    END /* LOOP FOR COMPARING ELEMENTS */
   /* A(LAST),..., A(N-1) ARE NOW OK */
   LAST := LAST -1
  END */ LOOP FOR CHECKING CURRENT SEQUENCE */
 END /* IF N>1 */
/* END PROC 'SORT' */
```

2.3.7 Parameter Passing by Reference

Procedures may communicate scalar (integer or string) results through the parameters passed to it by specifying that a parameter is a reference parameter. Logically, this means that the scalar variable itself is passed to the procedure rather than the value of the variable, just as for array parameters. Thus a procedure can then change the value of a variable in a CALL argument list.

A procedure declares a scalar parameter to be a reference parameter by means of the keyword REF . The following program illustrates the difference between normal parameter passing (by value) and reference parameters.

```
INT X, Z=2, M
PROC ADD1 (INT X, INT Y)
   X := X + Y
PROC ADD2 (REF INT X, INT Y)
   X := X + Y
PROC MAIN
   X := 3
   CALL ADD1 (X,Z)
   M := X
   CALL ADD2 (X,Z)
   M := X
   START MAIN
```

This program would set M to 3 then 5

Note that only variables (simple or subscripted) may be passed by reference. That is, netiher constants nor expressions (that do not consist of a variable only) may be passed by reference.

Functions may also have reference parameters.

2.3.8 Return Statement

The RETURN statement causes a return to the calling procedure or function. It may be any statement in a segment. The form

```
RETURN
```

is used for procedures, and the form

```
RETURN (<expression>)
```

is used for functions.

A function FIND which attempts to find a number in an array can be written to illustrate this statement:

```
INT FUNC FIND (INT NUMBER, INT ARRAY VALUES, INT SIZE)
INT I
I := 1
WHILE I <= SIZE
  DO
    IF VALUES (I) = NUMBER
      THEN /* FOUND */
        RETURN (I)
      ELSE
        I := I + 1
      END
  END
RETURN (0) /* NOT FOUND */
/* END FUNC 'FIND' */
```

Note that the last statement in a function need not be a RETURN (<expression>) if the structure of the function's statement list is such

that a return is always made from within the statement list.

2.4 Integer Expressions

An integer expression represents an integer value. An integer expression may be

1) a scalar integer variable (either a simple variable or a subscripted array variable);
2) an integer constant;
3) an integer function call;
4) an integer operation (such as + or -) where each operand may also be an expression;
5) an integer expression enclosed in parentheses.

2.4.1 Subscripted Array Variables

An array element is designated by following the array name with a subscript, enclosed in parentheses, whose value designates the number of the array element to be used. The subscript can be any integer expression.

For example

    A(3)

designates the 4th element of array A , while

    A(X + A(Y))

designates the element whose number is the value of X plus the value of the array element designated by A(Y) .

2.4.2 Function Calls

A function call has the form

    <identifier> {(<argument list>)}

where <identifier> is the name of the function. The rules for <argument list> are the same as for the CALL statement.

2.4.3 Constants

An integer constant may be designated by any sequence of decimal digits representing a valid non-negative integer value. Note that negative constants may usually be used where desired although such a constant

is formally viewed as the unary minus operation on a nonnegative constant in integer expressions.

For example, the following are valid SIMPL-M integer constants.

3

127

0

2.5 Basic Integer Operators

The operators described in this section all have integer expressions as operands and yield an integer result. Any arithmetic overflow that occurs in a calculation is ignored.

2.5.1 Arithmetic Operators

Addition (+) , subtraction (-) , and multiplication (*) are binary operators with the usual meaning. The integer divide (/) operator yields the integer quotient of its operands. Thus if the result of X/Y is Q , then X = Q*Y + R , where R is the remainder that was discarded in the integer divide.

The unary minus (-) operator yields the negative of its operand. Note that the expression -3 is formally viewed as the unary minus operation on the constant 3 although it would probably be logically (and equivalently) viewed as the constant "minue three" by the programmer. There is no unary plus operator in SIMPL-M.

2.5.2 Relational Operators

The relational operators are equal (=) , not equal (<>) , less than (<) , less than or equal (<=) , greater than (>) , and greater than or equal (>=) . The expression X=Y has value 1 if X and Y are equal, and value 0 otherwise. The remaining relational operators are similarly defined.

Note that the result of a relational operation always has value 1 or zero, depending on whether the relation is true or false, respectively. The relational operators can also be denoted by .EQ. , .NE. , .LT. , .LE. , .GT. , and .GE. , respectively.

2.5.3 Logical Operators

The logical operators .AND. , .OR. , and .NOT. are defined by:

X.AND.Y is 1 if both X and Y are nonzero, and is 0 otherwise X.OR.Y is 0 if both X and Y are zero, and is 1 otherwise .NOT.X is 1 if X is zero and is 0 otherwise

As is the case for relational operations, a logical operation always yields the result 1 or 0 .

Note that the logical operators yield the "natural" result. For example, the expression

X<Y .AND. Y<Z

will have the value 1 (i.e., will be "true") if Y is both greater than X and less than Z , and will have the value 0 (i.e., will be "false") otherwise.

2.5.4 Precedence

The precedence of the basic integer operations, from highest to lowest, is

.NOT. - (unary)	unary
* /	arithmetic
+ - (binary)	
= <> < > <= >=	relational
.AND.	logical
.OR.	

The order of evaluation between operators of equal precedence is left to right (except between unary operators, which is right to left).

As an example, the expression

- A + B + C * D

would be evaluated by

(1) negating the value of A

(2) adding the value of B to the result from (1)

(3) multiplying the value of C by the value of D

(4) adding the results from (2) and (3)

Parentheses may be used to alter the normal precedence. Thus (A+B)*C would cause the values of A and B to be added and the result to be multiplied by the value of C .

2.5.5 <u>Examples</u>

The following are examples of valid SIMPL-T expressions.

(1) X + Y/7 * 2

(2) X<3 .OR. X>8

(3) X>3 .AND. X+Y<10

(4) X + (X*(Y+1)<500)

For X=9 and Y=12 these expressions have the values

(1) 11

(2) 1

(3). 0

(4) 10

2.6 <u>Identifiers</u>

Identifiers (i.e., names) in SIMPL-M may be any string of letters or digits that begins with a letter. For usage in an identifier, the symbol $ is considered to be a letter. Identifiers are used to denote variables, arrays, procedures, functions, and other entities in a program. All identifiers used in a program must be declared.

There is no formal restriction on the length of identifiers. However identifiers may not cross the boundary of a source input record (e.g., card), so that there is an actual restriction to the length of an input record (e.g., 80 characters).

Certain reserved words (keywords) <u>may not</u> be used as identifiers in a SIMPL-M program. These keywords (such as IF , INT) are listed in Appendix III. Due to the special meaning given to these keywords, rather disastrous results may occur if a keyword is used as an identifier in a SIMPL-M program. This is especially true of keywords used in declarations (such as INT , ARRAY , PROC). The resulting diagnostics generated by the compiler may not be too helpful for usch an error, primarily because the programmer often

overlooks this type of error as a possible cause of the diagnostics.

Since many keywords are used for more specialized features of the SIMPL-M language, the list in Appendix III should be consulted before writing a SIMPL-M program.

2.7 Basic I/O

There is no pre-defined I/O for SIMPL-M (See Sections 1. and 1.2). Instead, the programmer can specify the start address of previously loaded I/O subroutines anywhere in memory above location 127. This is done by declaring EXTPROC and EXTFUNC as external and specifying the procedure start address as a parameter.

2.7.1 Function EXTFUNC (INT)

EXTFUNC performs a function call to the address specified by its parameter. The parameter is declared as a global initialized to the external subroutine address. EXTFUNC assumes the external procedure will return its argument in the 8080 accumulator. See Example 2.7.3.

2.7.2 Procedure EXTPROC (INT,INT)

EXTPROC performs a procedure call to the address specified by its first parameter and passes its second parameter to the subroutine in the accumulator. The first parameter is declared as a global initialized to the external subroutine address. See Example 2.7.3.

2.7.3 Example

```
EXT INT FUNC EXTFUNC(INT)
EXT PROC EXTPROC(INT,INT)

INT DOG=6, CAT=3
INT GETA=H↑0167↑/*GETA IS A PRE-LOADED ROUTINE*/
                  /*IN PROM WHICH INPUTS ONE HEX*/
                  /*DIGIT TO THE ACCUMULATOR*/
INT DIGOUT=H↑0143↑/*DIGOUT IS A PRE-LOADED ROUTINE*/
                  /*IN PROM WHICH OUTPUTS ONE HEX*/
                  /*DIGIT FROM THE ACCUMULATOR*/
```

```
PROC TESTIO
     CALL EXTPROC (DIGOUT,DOG+CAT+1)/*WRITE HEX A*/
     CAT := EXTFUNC(GETA)/*INPUT ONE HEX CHARACTER*/
     /*INPUT AND ECHO ONE HEX CHARACTER*/
     CALL EXTPROC(DIGOUT,EXTFUNC(GETA))
START TESTIO
```

3. Start Load Specification

The Intel 8080 machine code can be loaded starting at any address in memory. This is done by putting the following statement as the very first statement in the input deck.

```
INT STARTLOAD=(start address)
```

Where (start address) is an integer constant as described in Section 4.1. The startload address must be in the range Ø to 65,535. The default is address zero. The memory load is sequential and uninterrupted from the start load address.

Examples

```
INT STARTLOAD=H↑ϕ4ϕϕ↑
INT STARTLOAD=6ϕ
```

4. Additional Language Features

4.1 Bit Representation for Integer Constants

Integer constants may be specified in binary, octal, or hexadecimal, as well as decimal. However, these additional representations specify the bit pattern for the word in which the integer is stored, rather than the value of the integer. Thus a maximum of 7 bits may be specified for the INTEL 8080.

A bit representation consists of the letter B , O , or H , followed by the binary, octal, or hexadecimal, respectively, constant enclosed in up

arrows. (Embedded blanks are permitted.) For example, integer value 23 can also be specified by any of the following;

B↑10111↑ B↑010 111↑
O↑27↑
H↑17↑

Trailing zeros may conveniently be specified by ending the constant in quotes by the letter Z followed by the decimal number of zeros to be included. For example,

B↑11Z3↑ = B↑11000↑

A bit representation may occur anywhere in a SIMPL-M program that an integer constant may occur.

4.2 Bit Operators

4.2.1 Shift Operators

There are two shift operators in SIMPL-M: Left logical shift (.LL.), and right logical shift (.RL.). These are binary operators that are used in the form.

<integer expression> <shift operator> <shift count>

where <shift count> is an integer expression whose value is the number of bits to shift.

4.2.2 Bit Logical Operators

The bit logical operators complement (.C.), and (.A.), or (.V.), and exclusive or (.X.) also function the same as the corresponding INTEL 8080 hardware instructions. Examples are

.C. H↑1E↑ = H↑E1↑
B↑110101↑ .A. B↑11001↑ = B↑010001↑
B↑110101↑ .V. B↑011001↑ = B↑111101↑
B↑110101↑ .X. B↑011001↑ = B↑101100↑

4.2.3 Precedence

Bit complement (.C.) has the same precedence as the other unary operators but the binary bit operators have precedence over all other binary integer operators. Among the binary bit operators, the precedence (highest first) is

.LL.		.RL.	shift
.A.			
.V.	.X.		bit logical

4.3 Compiler Options

The following parameters may be listed on the SIMPL-T execute card to specify output.

S Print source deck

L Print Intel 8080 machine code

Q Print SIMPL quads

F Print cross-reference table.

See Appendix A for the proper format.

4.4 Source Program Print Commands

The following commands may be included anywhere in a SIMPL-M program to control print format.

/+ EJECT +/	Jump to top of next page.
/+ SKIP n +/	Skip n lines
/+ PRINTOFF +/ /+ PRINTON +/	Suppress printing between these two directives

Appendix I.

1. Compile a SIMPL-M program on the CDC Cyber 172.

```
Job Card
UPDATE,N.
REWIND,COMPILE.
REQUEST,COMPAS,*PF.
ATTACH,SIMPLT,SIMPLM.
SIMPLT,SLQ,COMPILE.  (See Note)
CATALOG,COMPAS,INTEL,RP=0.
7/8/9
*DECK Program Name
Program Deck
6/7/8/9
```

2. Generate a loader tape once a SIMPLM program is correct.

```
Operator Request Card
Job Card
REQUEST,PAPER,TP.
ATTACH(TAPE30,INTEL)
UVALIB(PUNPAPR,P1,,,,,P6)
6/7/8/9
```

Note: Leave off L and Q until program compiles to suppress printout of assembly code and quads.

Appendix II - Precedence of Operators

The SIMPL-M operators are listed below in order of precedence from highest to lowest.

.C. .NOT. - (unary)	unary
.RL. .LL.	shift
.A. .V. .X.	bit logical
* / + - (binary)	arithmetic
= <> < > <= >=	relational
.AND. .OR.	logical

Appendix III - Keywords

The following are reserved keywords and may not be used as identifiers in a SIMPL-M program.

ARRAY	DO	EXT	OF	RETURN
CALL	ELSE	FILE	OTHER	START
CASE	END	FUNC	PROC	STRING
CHAR	ENTRY	IF	REC	THEN
DEFINE	EXIT	INT	REF	WHILE
EXTPROC	EXTFUNC	STARTLOAD		

Appendix IV - UVA MODULAR MICROCOMPUTER SYSTEMS - BASIC MONITOR

Wesley E, McDonald, James H, Aylor

INTRODUCTION

The monitors which are available for the various microcomputer modular systems at the CSL have the same basic instruction set common to all. The methods of implementation are different for each, although this is transparent to the user. A description of the basic monitor is presented, followed by a discussion of each system's particular differences.

BASIC MONITOR INSTRUCTION SET

There are five instructions to the monitor, each consisting of one letter; the first letter of the desired function. These are;

1) Memory Display - M
2) Next Memory Display - N
3) Jump - J
4) Load Hex - L
5) Proceed - P

Any memory location may be altered by entering a colon and the new data in hexidecimal immediately following the monitor's response to either M or N. For example:

. M 0000 AF : CD

will result in location 0000 (which had AF_{16} in it) being changed to a CD_{16}.

The instruction formats are presented below. Any underlined items are characters input from the TTY. Xs denote Hexidecimal characters. The groups of four characters specify addresses and the groups of two characters specify data items.

.<u>M</u> <u>xxxx</u> xx <u>:</u> <u>xx</u>
.<u>M</u> <u>xxxx</u> xx <u>CR</u> LF
.<u>N</u> xxxx + 1 xx <u>:</u> <u>xx</u>
.<u>N</u> xxxx + 1 xx <u>CR</u> LF
.J <u>xxxx</u> <u>CR</u>
.<u>P</u> <u>CR</u>
.<u>L</u> (LOAD DATA according to format)

LOAD HEX

The load hex instruction has special provisions which allow communication with time share basic. If time share is not available, .L still will function as a paper tape loader. The paper tape format is:

LDR xoff CRLF	<CHECKSUM> = 4 Hex words, or 2 Hex bytes. The sum of all Hex data in line, e.g. BYTES+ADDR+DATA
;<BYTES><ADDR><DATA><CHECKSUM>CRLF	<BYTES>=# of bytes of hex code in line not to include the checksum bytes
;<BYTES><ADDR><DATA><CHECKSUM>CRLF	<ADDR> = starting address of data
;<BYTES><ADDR><DATA><CHECKSUM>CRLF	<DATA> consists of xx 2 hex words of data = 1 Hex Byte
; 00 CRLF	

The LDRXoff CRLF indicates to the loader that data will follow.
The semi-colon indicates that data follows immediately.
An end of record, specified by a ;00 terminates loading, and results in termination of the load phase. To run the loaded program, execute .J xxxx CR, where xxxx is the starting address of the loaded program.

Time Share Interface

If the time share interface is available (consisting only of a modem & telephone) the monitor provides communication with it through the .L instruction.

In order to prevent confusion, the monitor will not speak to timeshare unless in the .L routine. At all other times, the monitor is in a purely local mode, where commands and prompting are sent only to the system TTY.

Once in the .L routine, all TTY input is sent to the modem, and all modem input is sent to the TTY, allowing the use of the microcomputer system as a straight timesharing terminal. In the microcomputer system as a straight timesharing terminal. In the event that the LOADXX program on basic is executed, the monitor will automatically load the data coming from time share into RAM. The loaded program can then be executed through a reset and the .J instruction.

Program Break Points

Each monitor is arranged so that particular instructions in the machine code of the particular microprocessor will cause an entry into the monitor. Normally, such an entry preserves the stock and register integrity. When a .P is executed, the monitor automatically returns control to the calling program at the address of the previous break point.

Interrupts

All interrupts in the modular system jump through the monitor to a trap cell in RAM. This cell is initialized by the monitor to jump through the breakpoint. The RAM location should be initialized to a jump instruction to the interrupt service routine for special interrupt processing. Note, this is not an indirect jump. The program counter is loaded with the trap cell address and the processor begins execution at that point. Three bytes are allotted, so that a jump instruction can be executed.

SPECIAL FEATURES OF THE MODULAR MICROCOMPUTER

SYSTEMS BASIC MONITOR

INTEL 8080

Program Break Point:

The program break point is established through use of the RST 1 instruction. Control is transferred to the routine beginning at 10_B in ROM. All processor status and registers are preserved and printed out, in the following order:

XXXX	XX	XX	XX	XX	XX	XX	XX	XX
PC	A	B	C	D	E	H	L	PSW

Control is then transferred to the monitor, which responds with the prompting dot. Execution of .P will resume program execution.

MOTOROLA 6800

Program Break Point

The program break point is established through use of the SWI instruction. Control is transferred to the monitor which responds with a prompting dot. All processor status is preserved. Execution is continued through use of .P .

References.

1. Basili, Victor R., Albert J. Turner.
   SIMPL-T: A Structured Programming Language. Palladin House Publishers, Geneva, Ill. 61034, 1976.

2. Perry, John G., Jr., CDC 6000 SIMPL-T Compiler Internal Documentation. Unpublished Masters Thesis. University of Maryland, College Park, Md., 1976.

3. Intel Corporation. Intel 8080 Assembly Language Programming Manual. Intel Corp., Santa Clara, Ca., 95051, 1976.

4. McDonald, Wesley E., UVA Modular Microcomputer Systems Basic Monitor. Unpublished paper. The University of Virginia Electrical Engineering Lab., Charlottesville, Va. 22901, 1976.

5. Bladen, James B., SIMPL-M: A Cross-Compiler for the Intel 8080 Microcomputer. Unpublished Master's Thesis. The University of Virginia, Charlottesville, Va. 22901, 1977.

Appendix VII. CODGEN Listing — contact Capt Jim Bladen AFATL/DLMM Eglin AFB, Fla 32542 for listing

```
        /* THIS PROGRAM IS A CODE GENERATOR FOR SIMPLM WRITTEN BY JIM BLADEN  */
        /* OF THE COMPUTER SCIENCE DEPT OF THE UVA SCHOOL OF ENGINEERING.     */
        /* THE PROGRAM GENERATES INTEL 8080 MICROCOMPUTER MACHINE CODE IN A   */
        /* FORMAT ACCEPTABLE TO THE UVA IMPLEMENTATION OF THE INTEL 8080 LOAD-*/
        /* ER.  FURTHER DOCUMENTATION OF THIS PROGRAM IS AVAILABLE THROUGH THE*/
        /* COMPUTER SCIENCE DEPARTMENT 804-924-7201.                          */

        EXT INT UUPTNS, FGPTR, FCPTR
        EXT INT ARRAY SYMTAB
        EXT FILE S$DATA, S$QUAD, COMPAS
        EXT STRING FUNC NAME(INT,INT)
        EXT PROC WRITEC(FILE,STRING)
        EXT PROC CLOSE$$(FILE)

        INT     CONSTBIT = B↑1219↑,  INITBIT = B↑1231↑,   INTBIT = 1,
                ARYBIT = B↑1213↑,    PARMBIT = B↑1222↑,   REFBIT = B↑1225↑,
                PROCSW=0,            FUNCBIT = O↑17↑,     CHKSUM = 0,
                MAXCONST = 127,      MINCONST = -128,
                MEMPTR = -1,         MOTEMP = 1,          MOARRAY = 2,
                QUEPTR = 39,         DEQPTR = 39,         MOIMMED = B↑124↑,
                SAVEPTR = -1,        HALF = O↑777777↑,    LHALF = O↑7777776↑,
                ORGADDR = 0,         DUMMY,               AOPTION,
                FLUSH = 0,           FF = H↑FF↑,          FF00 = H↑FF00↑,
                FUNCSW = 0,          NUMWDS = 0,          FFFF = H↑FFFF↑,
                ID, AFLAG, A, BFLAG, B, RFLAG, R, /*CURRENT QUAD VALUES*/
                NID,NAFLAG,NA,NBFLAG,NB,NRFLAG,NR, /*NEXT QUAD VALUES*/
                LOPTION, ROPTION, COPTION, QOPTION   /*COMPILER OPTIONS*/

        INT ARRAY SAVE(100),  /*FORWARD ADDRESS STACK*/
                QUE(40),  /*POINTERS ARE QUEPTR,DEQPTR */
                NC(100)=(3,7(15),5(3),7(4),0,0,7,7,5,7,7,5,3,1,1,3,3,
                1,1,5,3,1(3),5,5,1,7,3(4),1,3,5,7,7,3,3,0)

        STRING ARRAY QUADS[6](58) = (
                ↑LINE  ↑, ↑=     ↑, ↑<>    ↑, ↑<     ↑, ↑<=    ↑, ↑>     ↑,
                ↑>=    ↑, ↑+     ↑, ↑-     ↑, ↑*     ↑, ↑/     ↑, ↑.A.   ↑,
                ↑.V.   ↑, ↑.X.   ↑, ↑.RL.  ↑, ↑.LL.  ↑, ↑NEG   ↑, ↑.C.   ↑,
                ↑.NOT. ↑, ↑.AND. ↑, ↑.OR.  ↑, ↑.RAS. ↑, ↑.LCS. ↑, ↑Q23   ↑,
                ↑Q24   ↑, ↑.CON. ↑, ↑INT   ↑, ↑.ABS. ↑, ↑MAX   ↑, ↑MIN   ↑,
                ↑:=    ↑, ↑IF    ↑, ↑ELSEIF↑, ↑ENDIF ↑, ↑WHILE ↑, ↑WHTEST↑,
                ↑ENDWH ↑, ↑INITCS↑, ↑CSTEST↑, ↑CASE  ↑, ↑CSEND ↑, ↑CSELSE↑,
                ↑ENDCS ↑, ↑CALL  ↑, ↑PARM  ↑, ↑ENDPRM↑, ↑ARYLOC↑, ↑START ↑,
                ↑RETURN↑, ↑EXIT  ↑, ↑PROC  ↑, ↑ENDPRC↑, ↑CONTRL↑, ↑Q53   ↑,
                ↑DEFPRT↑, ↑PARTOP↑, ↑TRACE ↑, ↑STAMNT↑)
```